Learning Apache Spark with Python

Wenqiang Feng

December 05, 2021

CONTENTS

Welcome to my **Learning Apache Spark with Python** note! In this note, you will learn a wide array of concepts about **PySpark** in Data Mining, Text Mining, Machine Learning and Deep Learning. The PDF version can be downloaded from HERE.

CHAPTER

ONE

PREFACE

1.1 About

1.1.1 About this note

This is a shared repository for Learning Apache Spark Notes. The PDF version can be downloaded from HERE. The first version was posted on Github in ChenFeng ([Feng2017]). This shared repository mainly contains the self-learning and self-teaching notes from Wenqiang during his IMA Data Science Fellowship. The reader is referred to the repository https://github.com/runawayhorse001/LearningApacheSpark for more details about the `dataset` and the `.ipynb` files.

In this repository, I try to use the detailed demo code and examples to show how to use each main functions. If you find your work wasn't cited in this note, please feel free to let me know.

Although I am by no means an data mining programming and Big Data expert, I decided that it would be useful for me to share what I learned about PySpark programming in the form of easy tutorials with detailed example. I hope those tutorials will be a valuable tool for your studies.

The tutorials assume that the reader has a preliminary knowledge of programming and Linux. And this document is generated automatically by using sphinx.

1.1.2 About the author

- **Wenqiang Feng**
  - Director of Data Science and PhD in Mathematics
  - University of Tennessee at Knoxville
  - Email: von198@gmail.com

- **Biography**

  Wenqiang Feng is the Director of Data Science at American Express (AMEX). Prior to his time at AMEX, Dr. Feng was a Sr. Data Scientist in Machine Learning Lab, H&R Block. Before joining Block, Dr. Feng was a Data Scientist at Applied Analytics Group, DST (now SS&C). Dr. Feng's responsibilities include providing clients with access to cutting-edge skills and technologies, including Big Data analytic solutions, advanced analytic and data enhancement techniques and modeling.

Dr. Feng has deep analytic expertise in data mining, analytic systems, machine learning algorithms, business intelligence, and applying Big Data tools to strategically solve industry problems in a cross-functional business. Before joining DST, Dr. Feng was an IMA Data Science Fellow at The Institute for Mathematics and its Applications (IMA) at the University of Minnesota. While there, he helped startup companies make marketing decisions based on deep predictive analytics.

Dr. Feng graduated from University of Tennessee, Knoxville, with Ph.D. in Computational Mathematics and Master's degree in Statistics. He also holds Master's degree in Computational Mathematics from Missouri University of Science and Technology (MST) and Master's degree in Applied Mathematics from the University of Science and Technology of China (USTC).

- **Declaration**

  The work of Wenqiang Feng was supported by the IMA, while working at IMA. However, any opinion, finding, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the IMA, UTK, DST, HR & Block and AMEX.

1.2 Motivation for this tutorial

I was motivated by the IMA Data Science Fellowship project to learn PySpark. After that I was impressed and attracted by the PySpark. And I foud that:

1. It is no exaggeration to say that Spark is the most powerful Bigdata tool.
2. However, I still found that learning Spark was a difficult process. I have to Google it and identify which one is true. And it was hard to find detailed examples which I can easily learned the full process in one file.
3. Good sources are expensive for a graduate student.

1.3 Copyright notice and license info

1.4 Acknowledgement

At here, I would like to thank Ming Chen, Jian Sun and Zhongbo Li at the University of Tennessee at Knoxville for the valuable discussion and thank the generous anonymous authors for providing the detailed solutions and source code on the internet. Without those help, this repository would not have been possible to be made. Wenqiang also would like to thank the Institute for Mathematics and Its Applications (IMA) at University of Minnesota, Twin Cities for support during his IMA Data Scientist Fellow visit and thank TAN THIAM HUAT and Mark Rabins for finding the typos.

A special thank you goes to Dr. Haiping Lu, Lecturer in Machine Learning at Department of Computer Science, University of Sheffield, for recommending and heavily using my tutorial in his teaching class and for the valuable suggestions.

1.5 Feedback and suggestions

Your comments and suggestions are highly appreciated. I am more than happy to receive corrections, suggestions or feedbacks through email (von198@gmail.com) for improvements.

CHAPTER

TWO

WHY SPARK WITH PYTHON ?

Chinese proverb

Sharpening the knife longer can make it easier to hack the firewood – old Chinese proverb

I want to answer this question from the following two parts:

2.1 Why Spark?

I think the following four main reasons from Apache Spark™ official website are good enough to convince you to use Spark.

1. Speed

   Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

   Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.

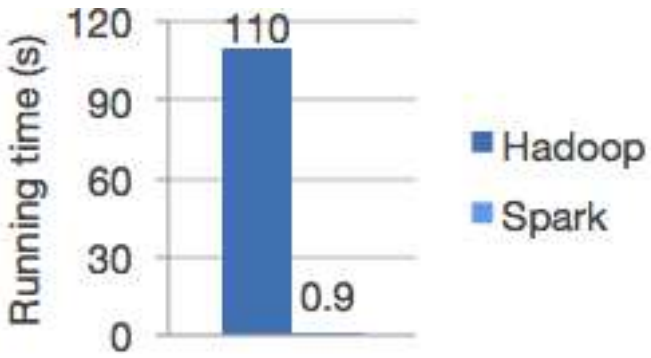


Fig. 1: Logistic regression in Hadoop and Spark

2. Ease of Use

   Write applications quickly in Java, Scala, Python, R.

Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it interactively from the Scala, Python and R shells.

3. Generality

   Combine SQL, streaming, and complex analytics.

   Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application.

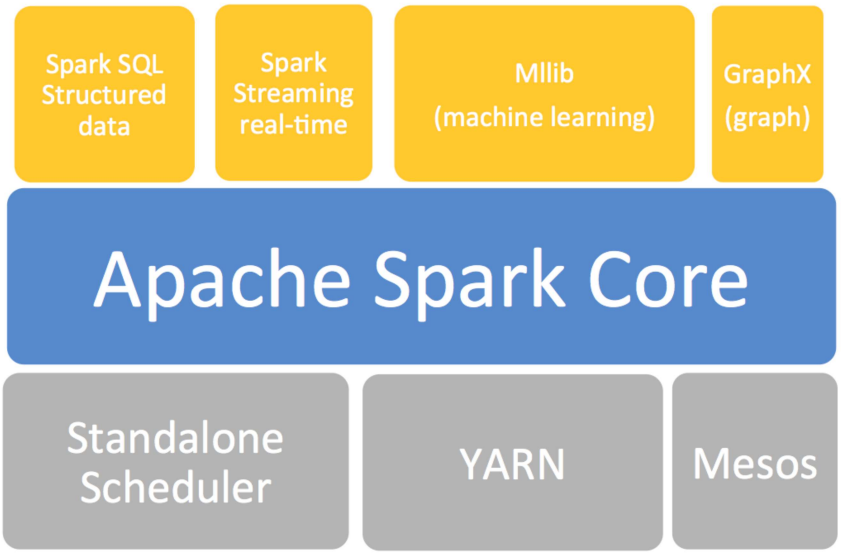


Fig. 2: The Spark stack

4. Runs Everywhere

   Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

Fig. 3: The Spark platform

2.2 Why Spark with Python (PySpark)?

No matter you like it or not, Python has been one of the most popular programming languages.

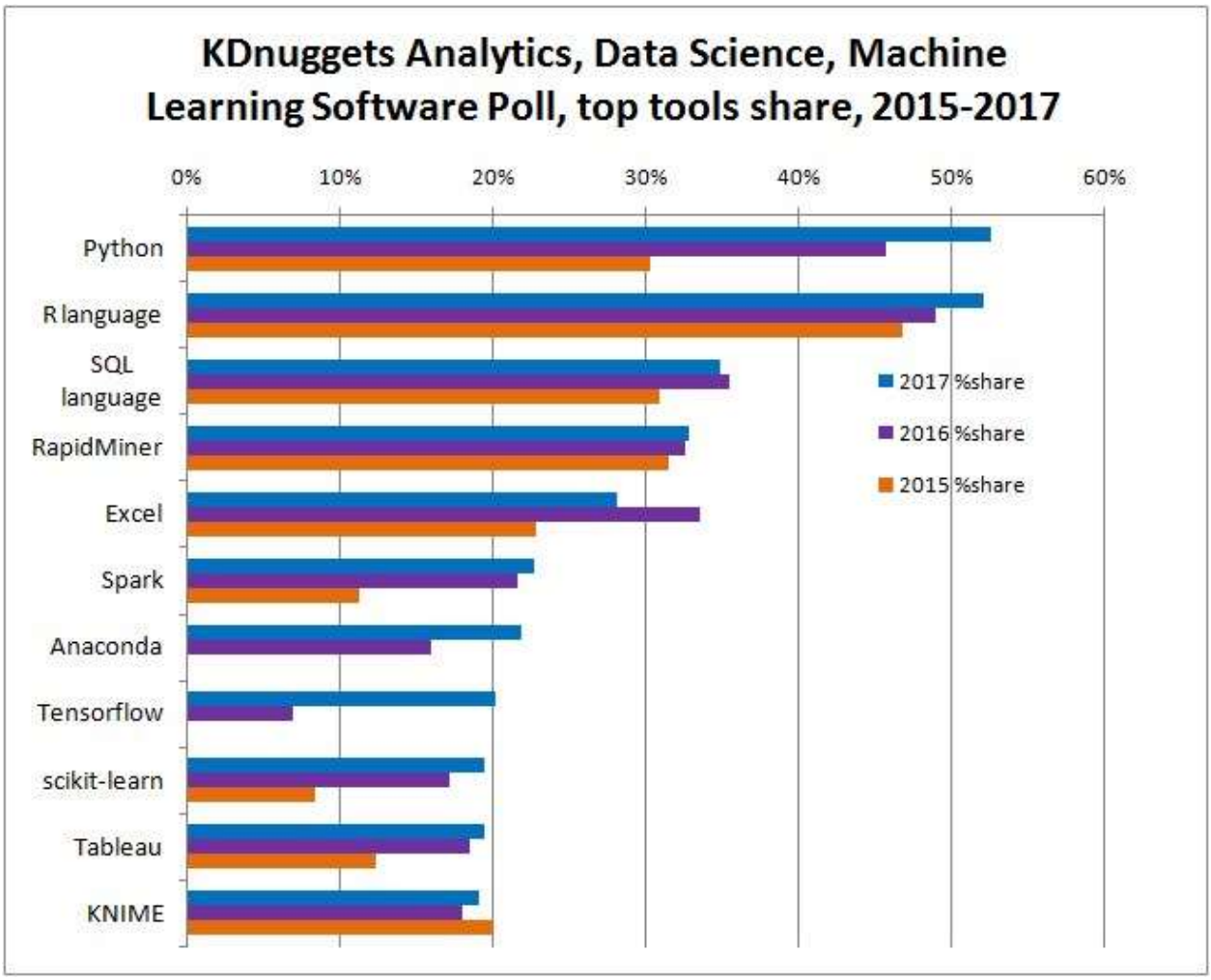


Fig. 4: KDnuggets Analytics/Data Science 2017 Software Poll from kdnuggets.

CHAPTER

THREE

CONFIGURE RUNNING PLATFORM

Chinese proverb

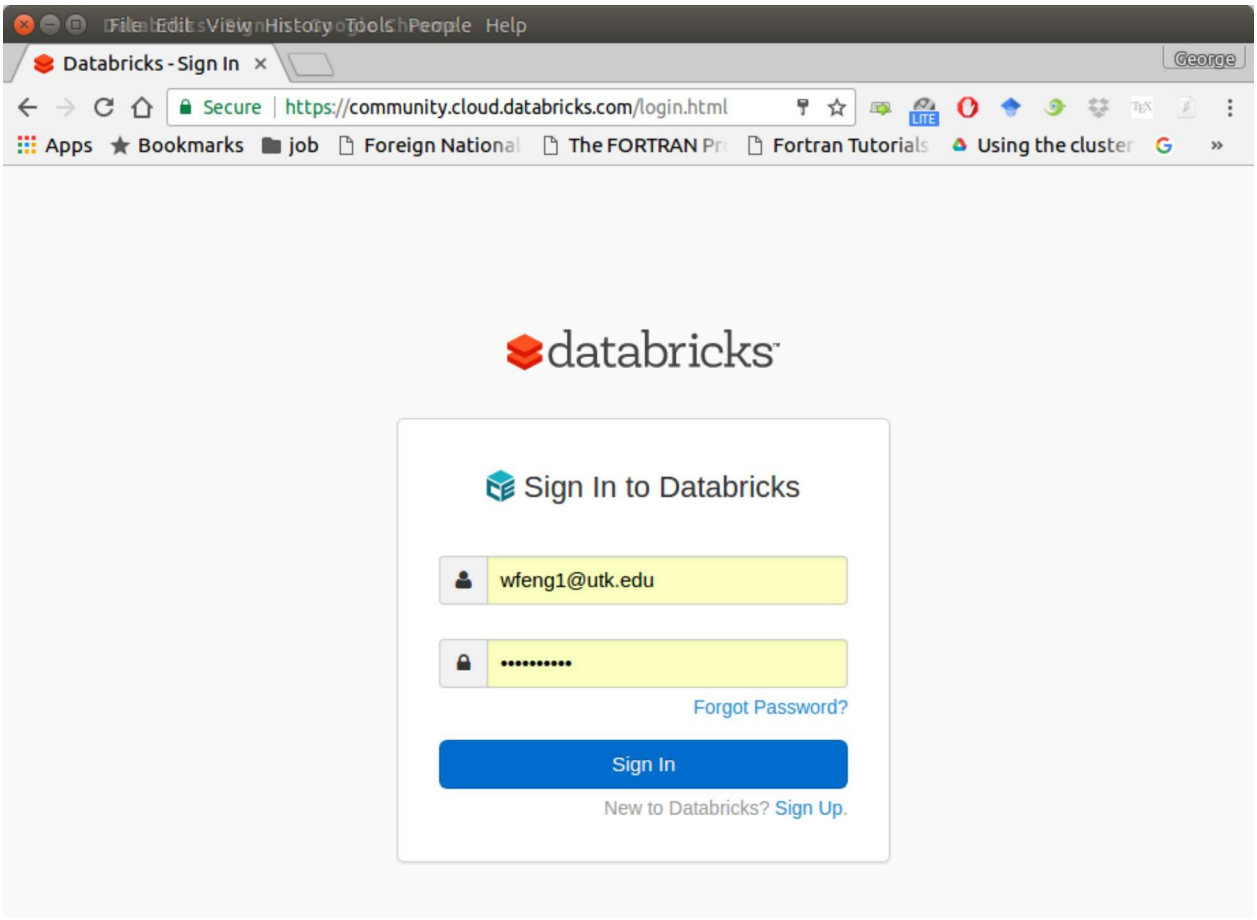
Good tools are prerequisite to the successful execution of a job. – old Chinese proverb

A good programming platform can save you lots of troubles and time. Herein I will only present how to install my favorite programming platform and only show the easiest way which I know to set it up on Linux system. If you want to install on the other operator system, you can Google it. In this section, you may learn how to set up Pyspark on the corresponding programming platform and package.

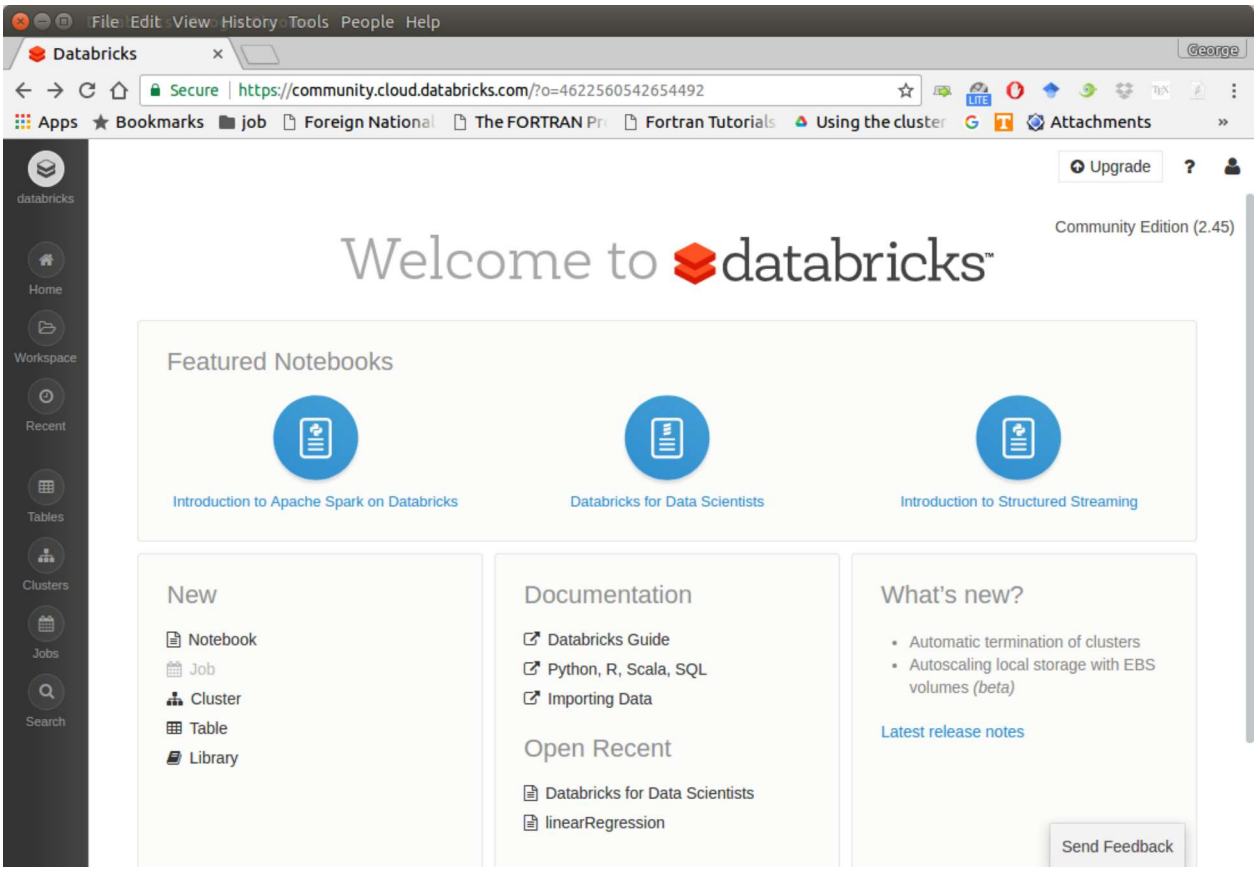
3.1 Run on Databricks Community Cloud

If you don't have any experience with Linux or Unix operator system, I would love to recommend you to use Spark on Databricks Community Cloud. Since you do not need to setup the Spark and it's totally **free** for Community Edition. Please follow the steps listed below.

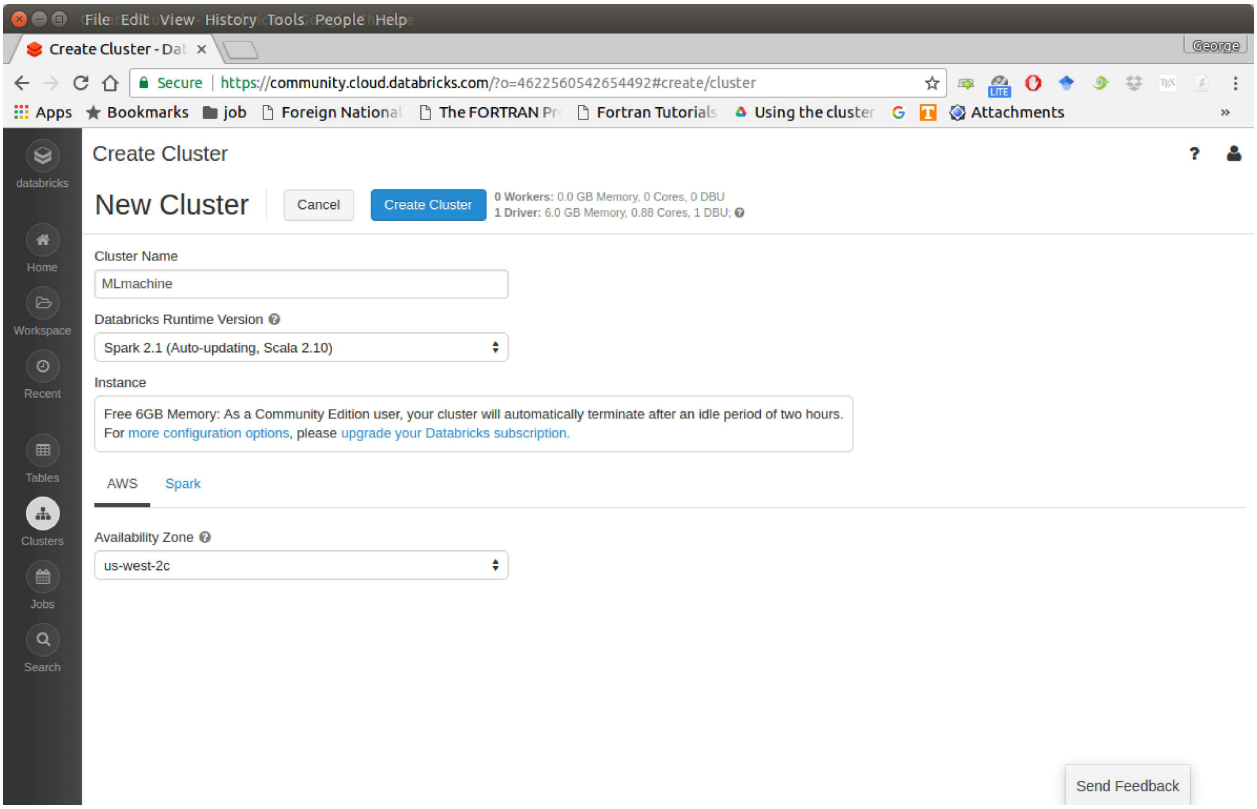
1. Sign up a account at: https://community.cloud.databricks.com/login.html



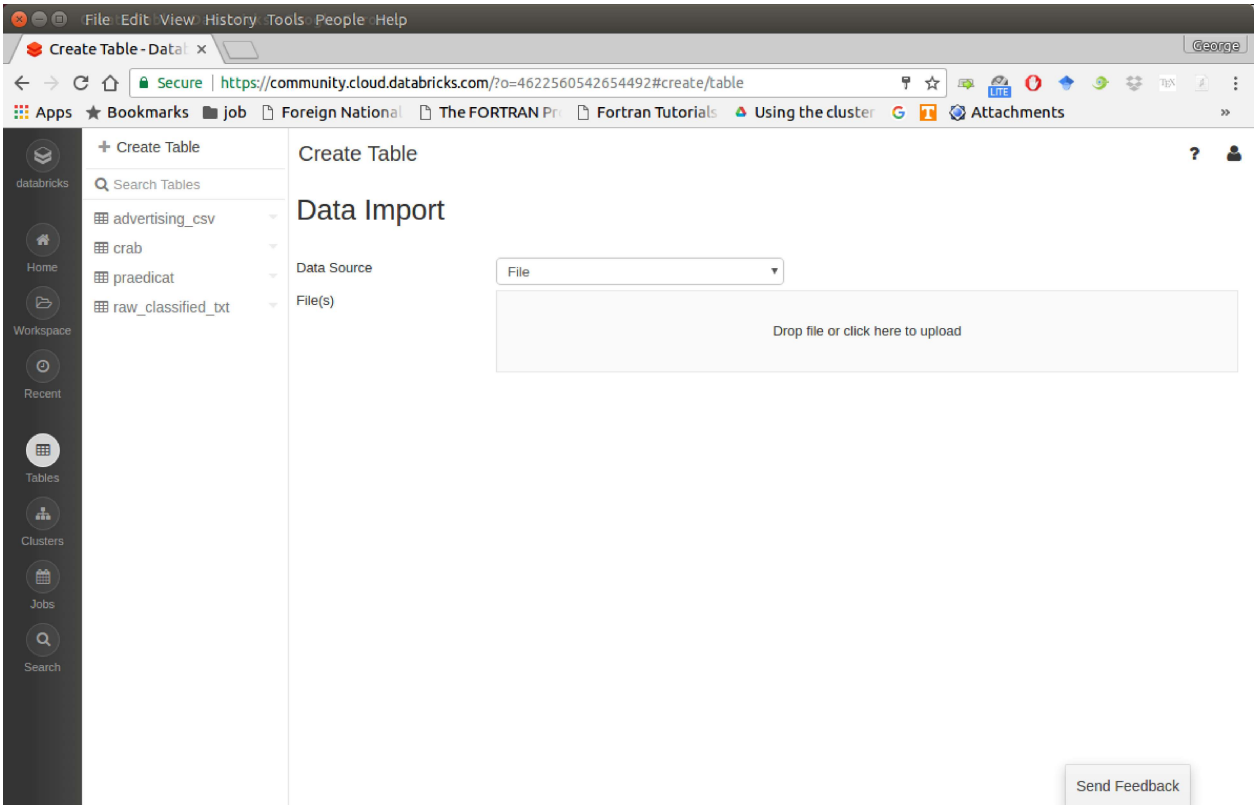
2. Sign in with your account, then you can creat your cluster(machine), table(dataset) and notebook(code).

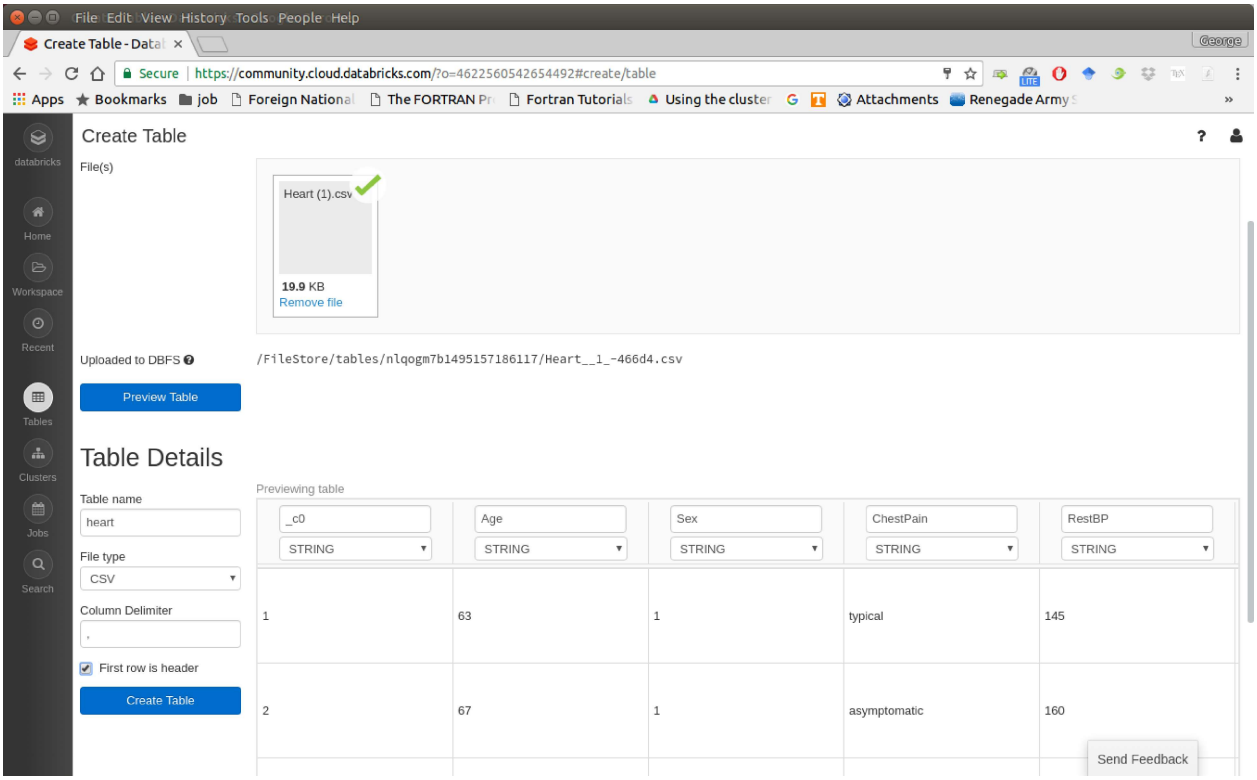


3. Create your cluster where your code will run



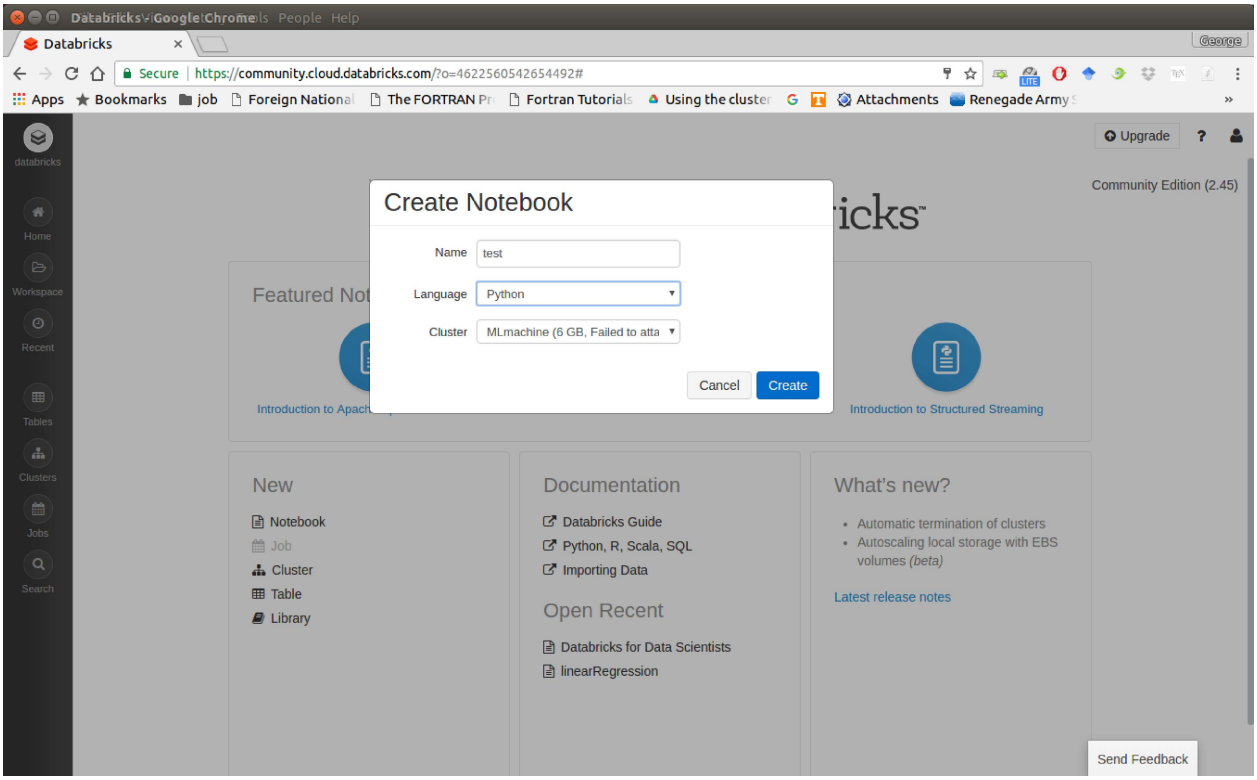
4. Import your dataset

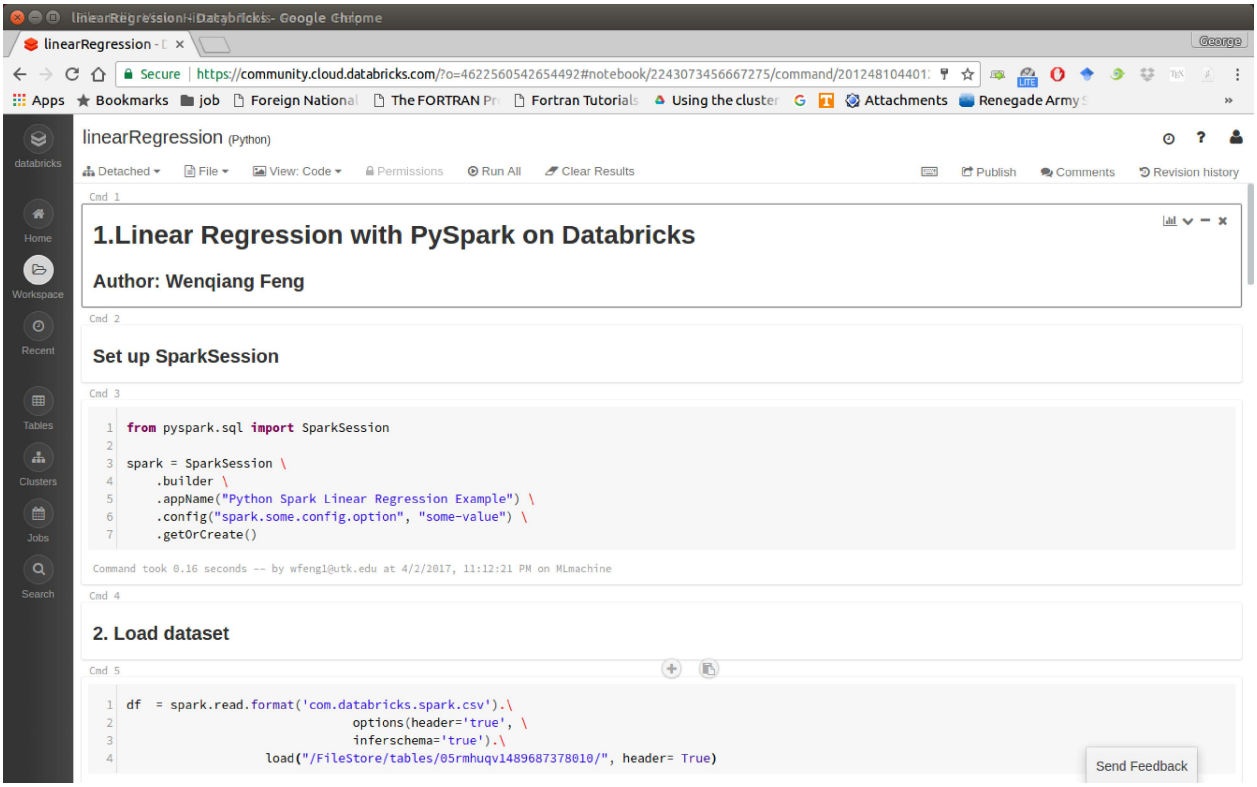

File Edit View History Tools People Help
Create Table - Data
George
Secure | https://community.cloud.databricks.com/?o=4622560542654492#create/table
Apps Bookmarks job Foreign National The FORTRAN Pr Fortran Tutorials Using the cluster Attachments
databricks
Home
Workspace
Recent
Tables
Clusters
Jobs
Search
Create Table
Search Tables
advertising_csv
crab
praedicat
raw_classified_txt
Create Table
Data Import
Data Source
File
File(s)
Drop file or click here to upload
Send Feedback


File Edit View History Tools People Help
Create Table - Data
George
Secure | https://community.cloud.databricks.com/?o=4622560542654492#create/table
Apps Bookmarks job Foreign National The FORTRAN Pr Fortran Tutorials Using the cluster Attachments Renegade Army
Create Table
File(s)
Heart (1).csv
19.9 KB
Remove file
Uploaded to DBFS
/FileStore/tables/nlqogm7b1495157186117/Heart__1_-466d4.csv
Preview Table
Table Details
Table name
heart
File type
CSV
Column Delimiter
,
First row is header
Create Table
Previewing table
_c0 Age Sex ChestPain RestBP
STRING STRING STRING STRING STRING
1 63 1 typical 145
2 67 1 asymptomatic 160
Send Feedback

Note: You need to save the path which appears at Uploaded to DBFS: /FileStore/tables/05rmhuqv1489687378010/. Since we will use this path to load the dataset.

5. Create your notebook





After finishing the above 5 steps, you are ready to run your Spark code on Databricks Community Cloud. I will run all the following demos on Databricks Community Cloud. Hopefully, when you run the demo code, you will get the following results:

```
+---+-----+-----+---------+-----+
|_c0|   TV|Radio|Newspaper|Sales|
+---+-----+-----+---------+-----+
|  1|230.1| 37.8|     69.2| 22.1|
|  2| 44.5| 39.3|     45.1| 10.4|
|  3| 17.2| 45.9|     69.3|  9.3|
|  4|151.5| 41.3|     58.5| 18.5|
|  5|180.8| 10.8|     58.4| 12.9|
+---+-----+-----+---------+-----+
only showing top 5 rows

root
 |-- _c0: integer (nullable = true)
 |-- TV: double (nullable = true)
 |-- Radio: double (nullable = true)
 |-- Newspaper: double (nullable = true)
 |-- Sales: double (nullable = true)
```

3.2 Configure Spark on Mac and Ubuntu

3.2.1 Installing Prerequisites

I will strongly recommend you to install Anaconda, since it contains most of the prerequisites and support multiple Operator Systems.

1. **Install Python**

Go to Ubuntu Software Center and follow the following steps:

a. Open Ubuntu Software Center

b. Search for python

c. And click Install

Or Open your terminal and using the following command:

```
sudo apt-get install build-essential checkinstall
sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev
                libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev
sudo apt-get install python
sudo easy_install pip
sudo pip install ipython
```

3.2.2 Install Java

Java is used by many other softwares. So it is quite possible that you have already installed it. You can by using the following command in Command Prompt:

```
java -version
```

Otherwise, you can follow the steps in How do I install Java for my Mac? to install java on Mac and use the following command in Command Prompt to install on Ubuntu:

```
sudo apt-add-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
```

3.2.3 Install Java SE Runtime Environment

I installed ORACLE Java JDK.

Warning: Installing Java and Java SE Runtime Environment steps are very important, since Spark is a domain-specific language written in Java.

You can check if your Java is available and find it's version by using the following command in Command Prompt:

```
java -version
```

If your Java is installed successfully, you will get the similar results as follows:

```
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
```

3.2.4 Install Apache Spark

Actually, the Pre-build version doesn't need installation. You can use it when you unpack it.

a. Download: You can get the Pre-built Apache Spark™ from Download Apache Spark™.

b. Unpack: Unpack the Apache Spark™ to the path where you want to install the Spark.

c. Test: Test the Prerequisites: change the direction `spark-#.#.#-bin-hadoop#.#/bin` and run

```
./pyspark
```

```
Python 2.7.13 |Anaconda 4.4.0 (x86_64)| (default, Dec 20 2016,␣
↪23:05:08)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more␣
↪information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.
↪properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR,
use setLogLevel(newLevel).
17/08/30 13:30:12 WARN NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where␣
↪applicable
17/08/30 13:30:17 WARN ObjectStore: Failed to get database global_
↪temp,
returning NoSuchObjectException
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.1.1
      /_/

Using Python version 2.7.13 (default, Dec 20 2016 23:05:08)
SparkSession available as 'spark'.
```

3.2.5 Configure the Spark

a. **Mac Operator System:** open your `bash_profile` in Terminal

```
vim ~/.bash_profile
```

And add the following lines to your `bash_profile` (remember to change the path)

```
# add for spark
export SPARK_HOME=your_spark_installation_path
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
export PATH=$PATH:$SPARK_HOME/bin
export PYSPARK_DRIVER_PYTHON="jupyter"
export PYSPARK_DRIVER_PYTHON_OPTS="notebook"
```

At last, remember to source your `bash_profile`

```
source ~/.bash_profile
```

b. **Ubuntu Operator Sysytem:** open your `bashrc` in Terminal

```
vim ~/.bashrc
```

And add the following lines to your `bashrc` (remember to change the path)

```
# add for spark
export SPARK_HOME=your_spark_installation_path
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
export PATH=$PATH:$SPARK_HOME/bin
export PYSPARK_DRIVE_PYTHON="jupyter"
export PYSPARK_DRIVE_PYTHON_OPTS="notebook"
```

At last, remember to source your `bashrc`

```
source ~/.bashrc
```

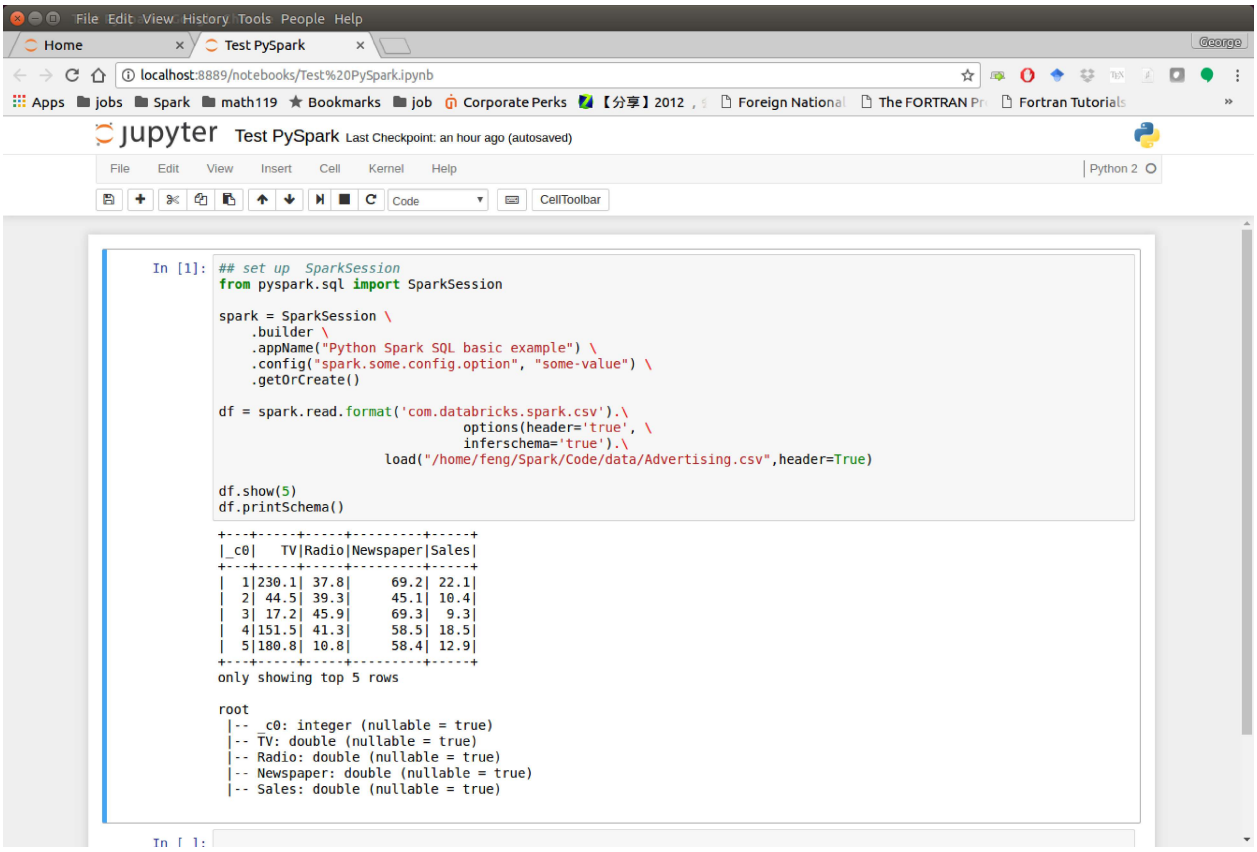
3.3 Configure Spark on Windows

Installing open source software on Windows is always a nightmare for me. Thanks for Deelesh Mandloi. You can follow the detailed procedures in the blog Getting Started with PySpark on Windows to install the Apache Spark™ on your Windows Operator System.

3.4 PySpark With Text Editor or IDE

3.4.1 PySpark With Jupyter Notebook

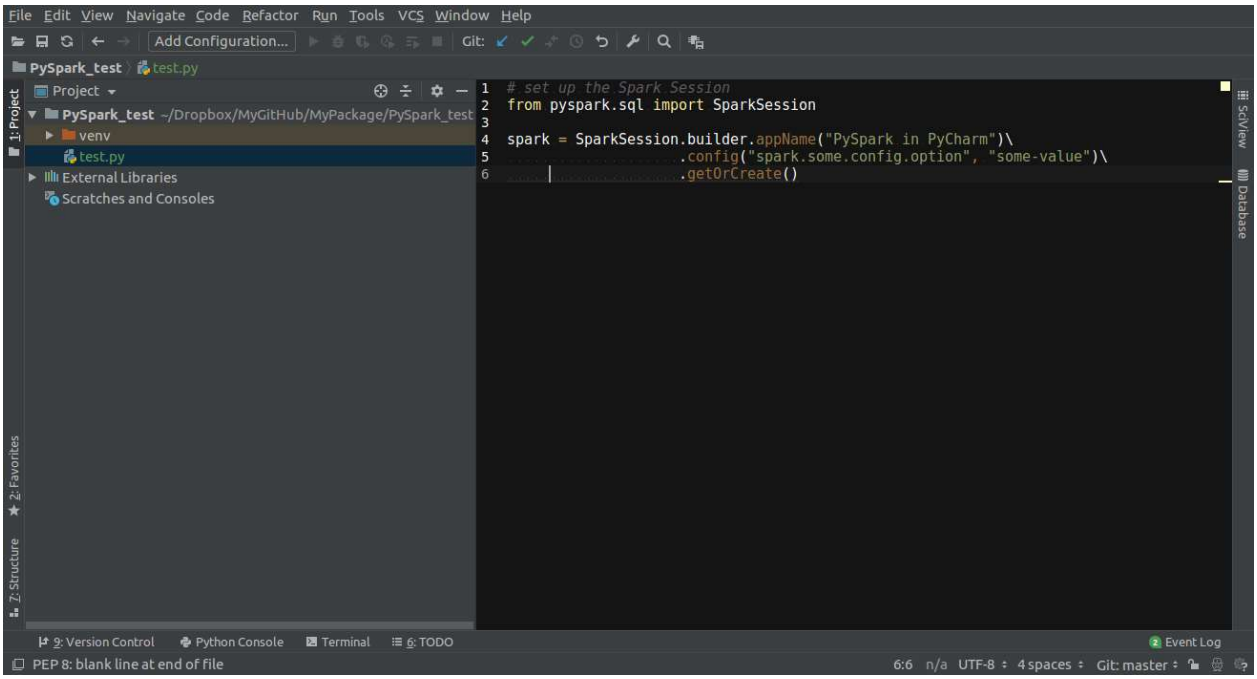
After you finishing the above setup steps in *Configure Spark on Mac and Ubuntu*, then you should be good to write and run your PySpark Code in Jupyter notebook.



3.4.2 PySpark With PyCharm

After you finishing the above setup steps in *Configure Spark on Mac and Ubuntu*, then you should be good to add the PySpark to your PyCharm project.

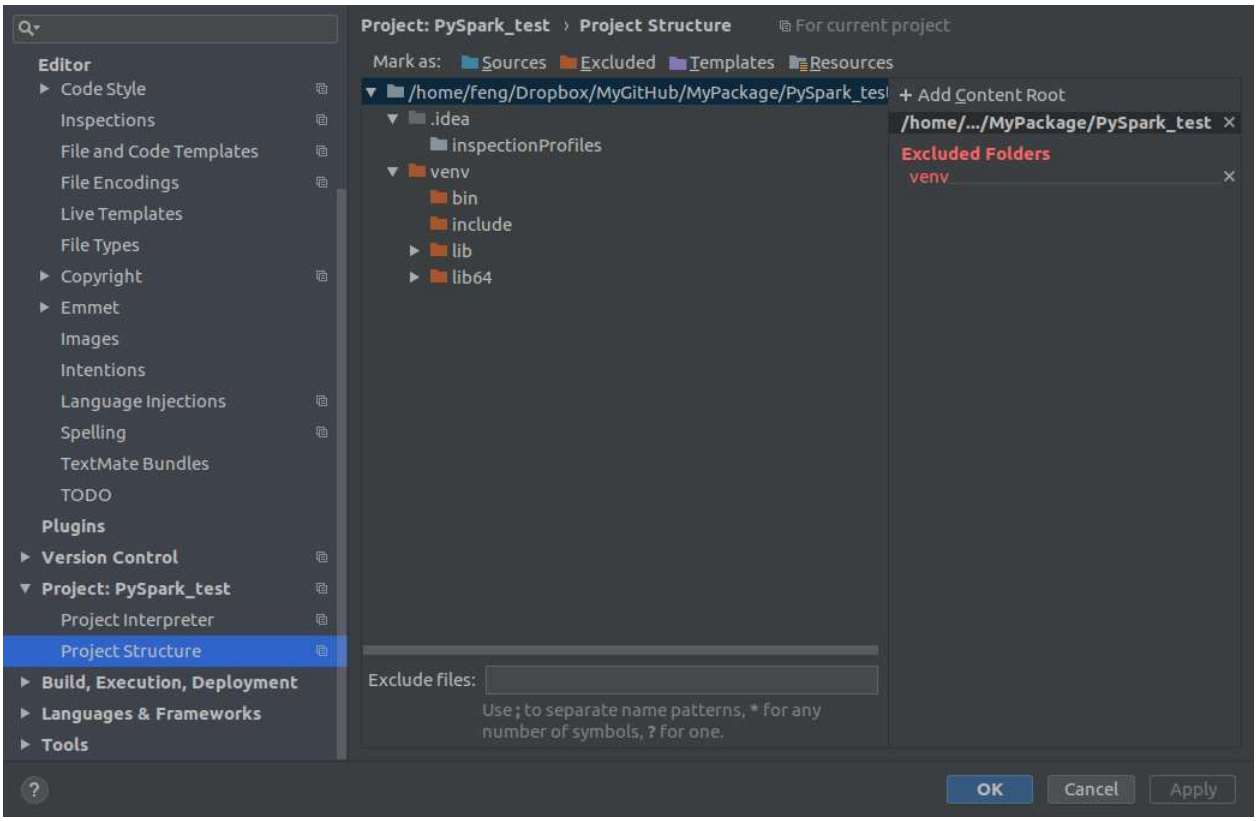
1. Create a new PyCharm project



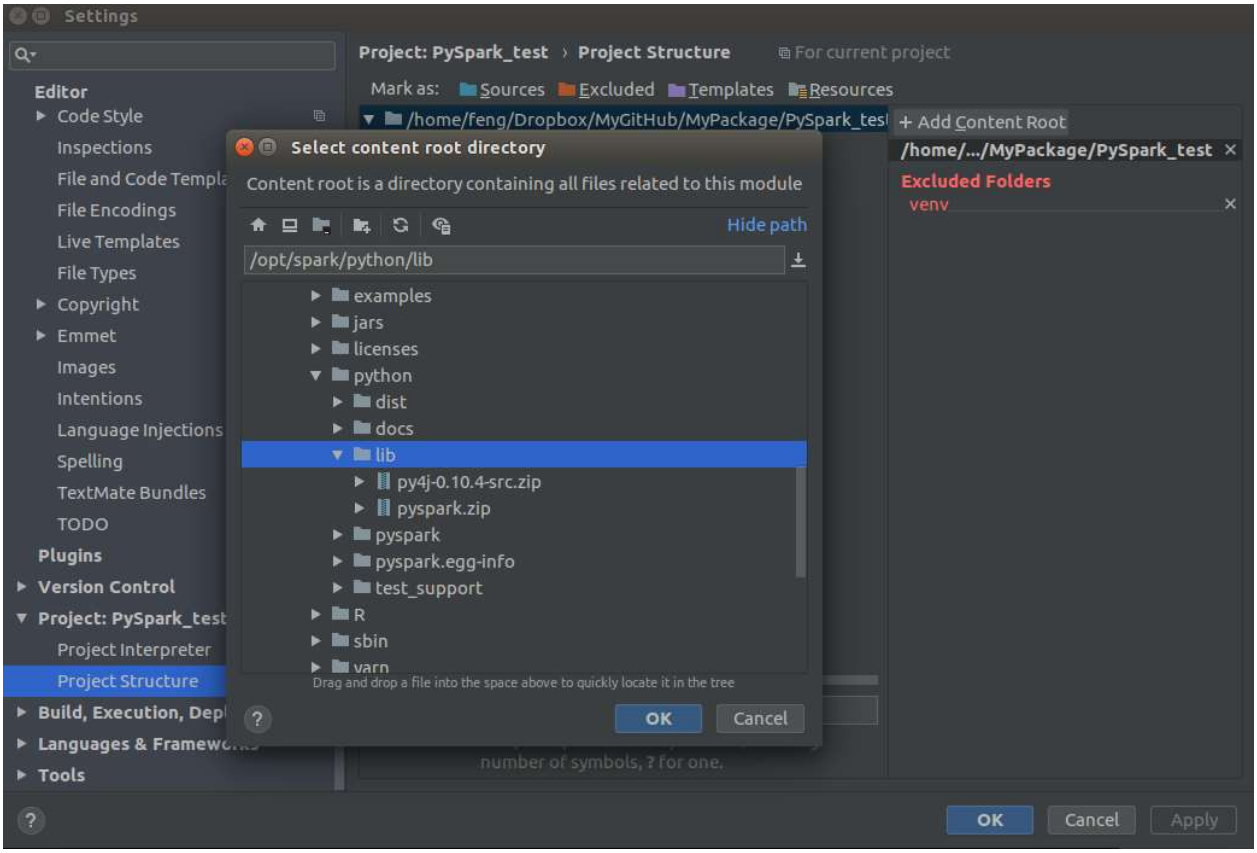
2. Go to Project Structure

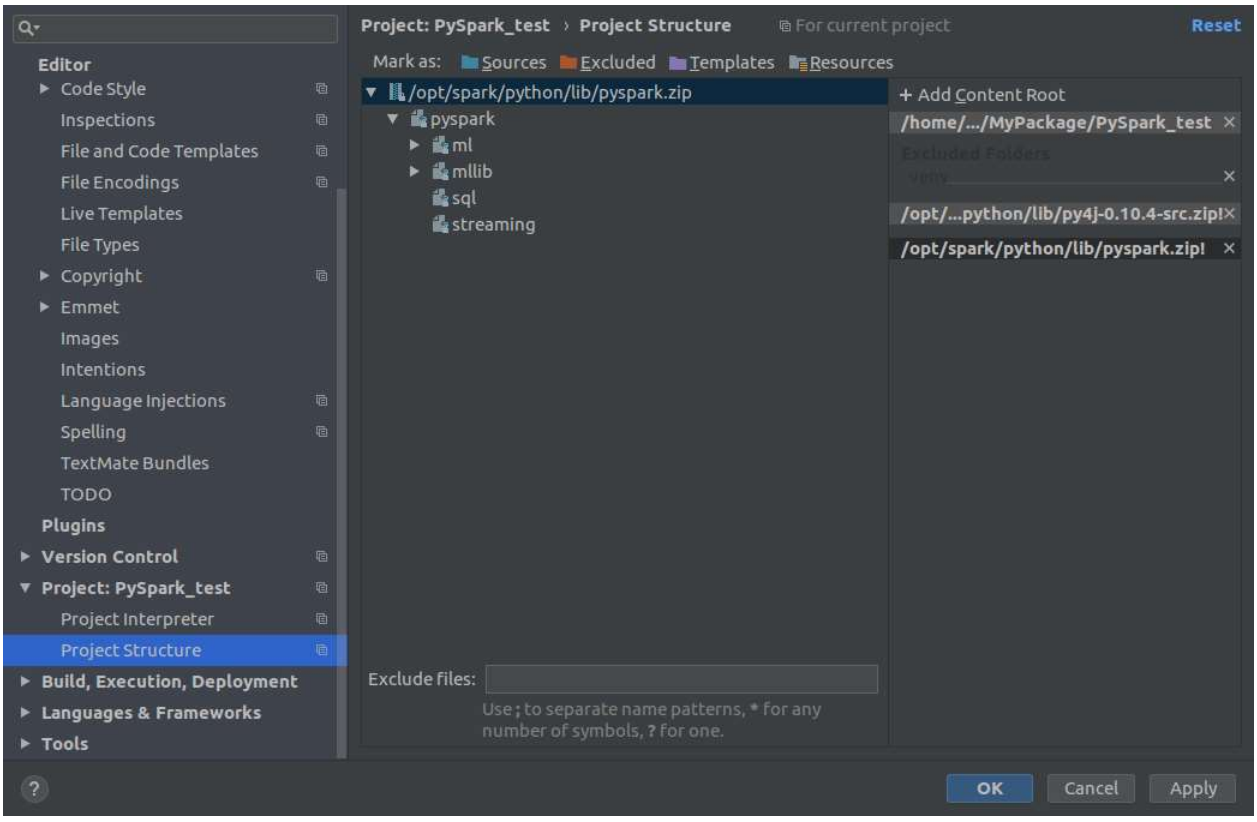
   Option 1: File -> Settings -> Project: -> Project Structure

   Option 2: PyCharm -> Preferences -> Project: -> Project Structure

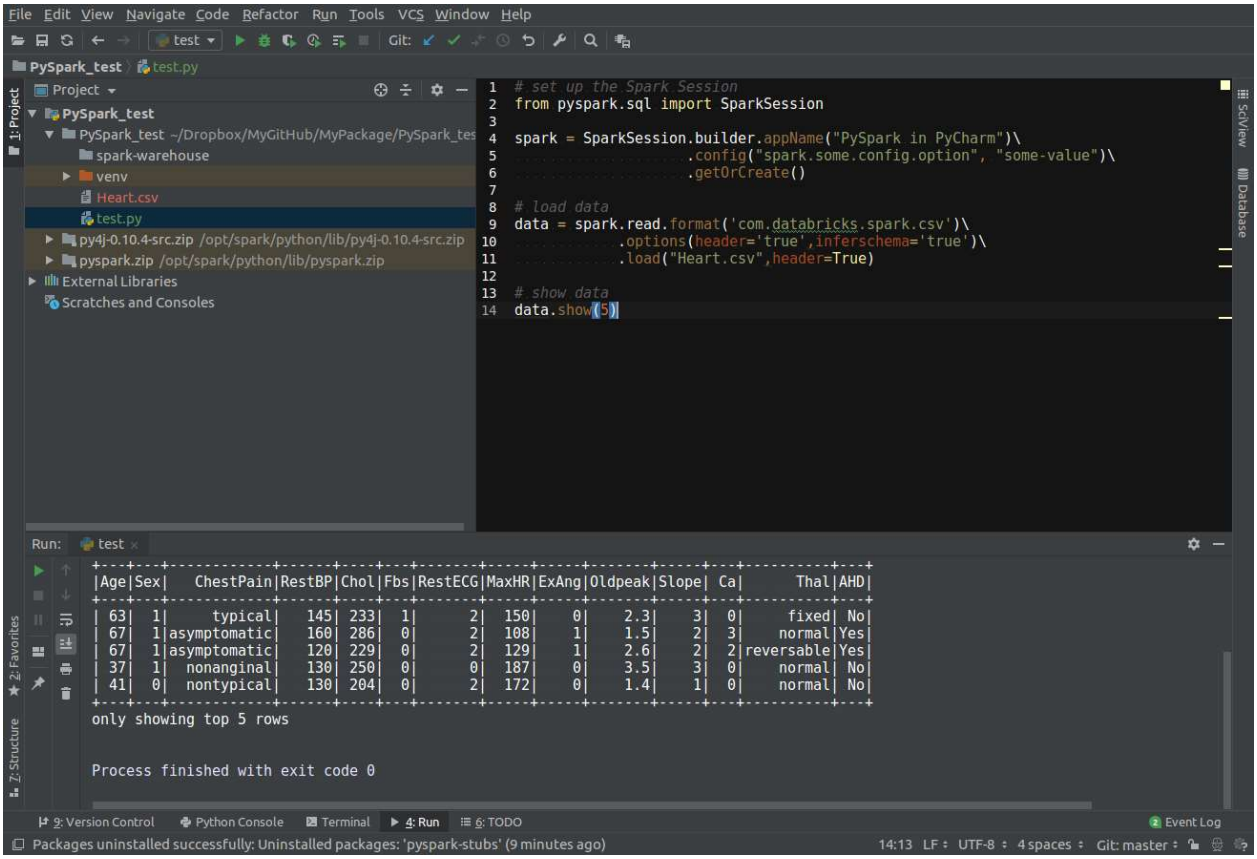


3. Add Content Root: all `ZIP` files from $SPARK_HOME/python/lib


Settings
Project: PySpark_test › Project Structure
For current project
Mark as: Sources Excluded Templates Resources
+ Add Content Root
/home/.../MyPackage/PySpark_test
Excluded Folders
venv
Select content root directory
Content root is a directory containing all files related to this module
Hide path
/opt/spark/python/lib
examples
jars
licenses
python
dist
docs
lib
py4j-0.10.4-src.zip
pyspark.zip
pyspark
pyspark.egg-info
test_support
R
sbin
Drag and drop a file into the space above to quickly locate it in the tree
OK
Cancel
Editor
Code Style
Inspections
File and Code Templates
File Encodings
Live Templates
File Types
Copyright
Emmet
Images
Intentions
Language Injections
Spelling
TextMate Bundles
TODO
Plugins
Version Control
Project: PySpark_test
Project Interpreter
Project Structure
Build, Execution, Deployment
Languages & Frameworks
Tools
Apply

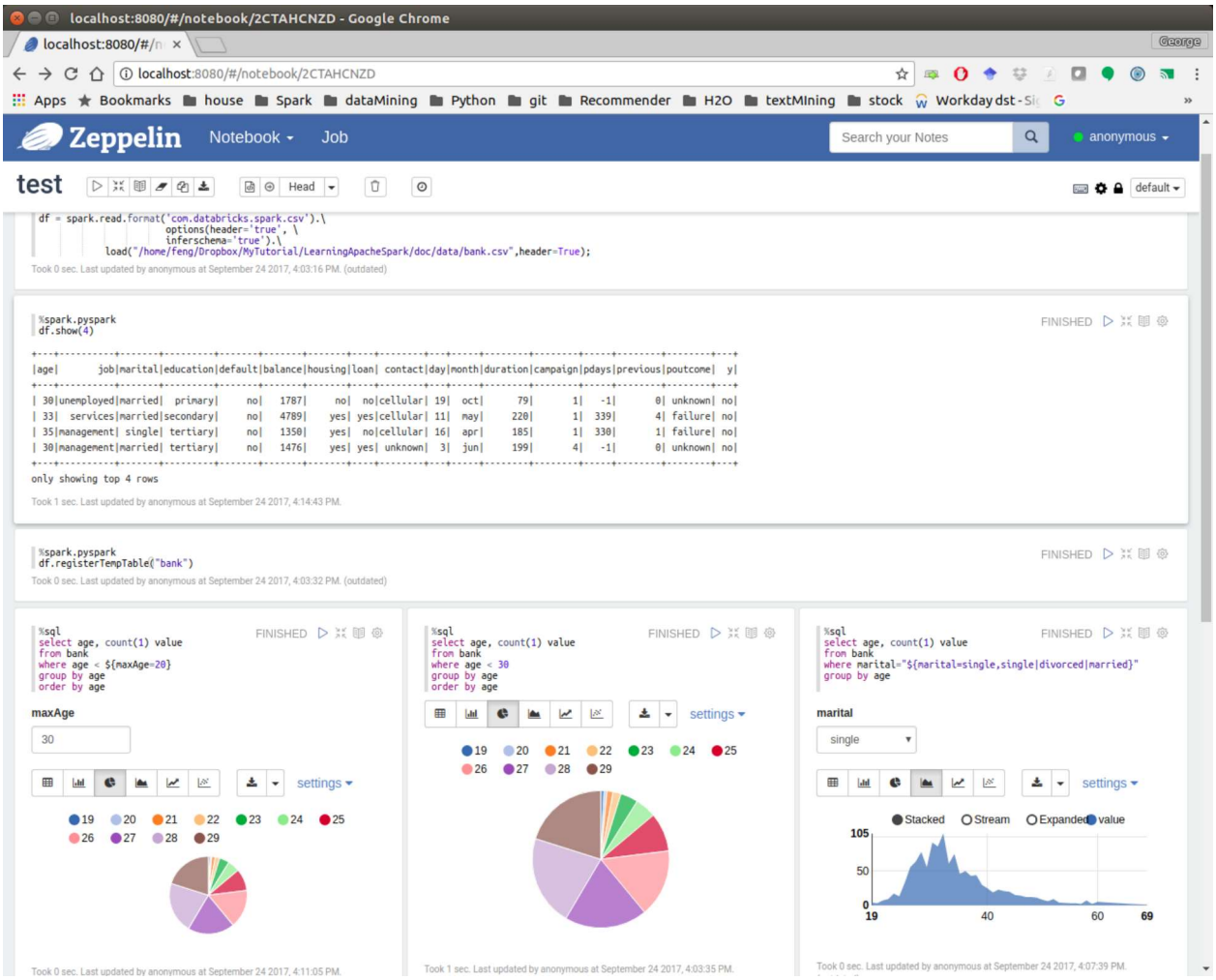

Project: PySpark_test › Project Structure
For current project
Reset
Mark as: Sources Excluded Templates Resources
/opt/spark/python/lib/pyspark.zip
pyspark
ml
mllib
sql
streaming
+ Add Content Root
/home/.../MyPackage/PySpark_test
/opt/...python/lib/py4j-0.10.4-src.zip!
/opt/spark/python/lib/pyspark.zip!
Exclude files:
Use ; to separate name patterns, * for any number of symbols, ? for one.
OK
Cancel
Apply

4. Run your script



3.4.3 PySpark With Apache Zeppelin

After you finishing the above setup steps in *Configure Spark on Mac and Ubuntu*, then you should be good to write and run your PySpark Code in Apache Zeppelin.

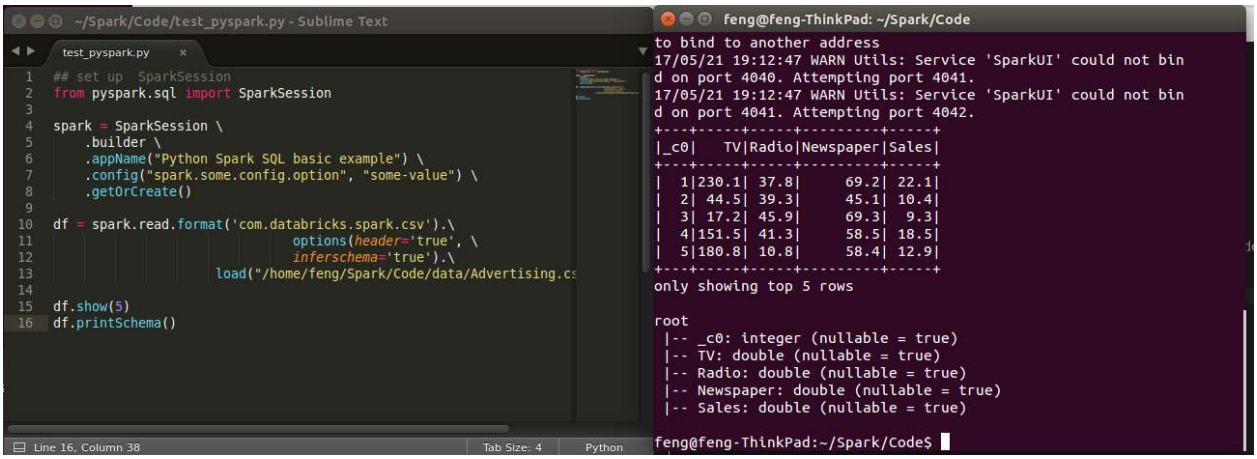


3.4.4 PySpark With Sublime Text

After you finishing the above setup steps in *Configure Spark on Mac and Ubuntu*, then you should be good to use Sublime Text to write your PySpark Code and run your code as a normal python code in Terminal.

```
python test_pyspark.py
```

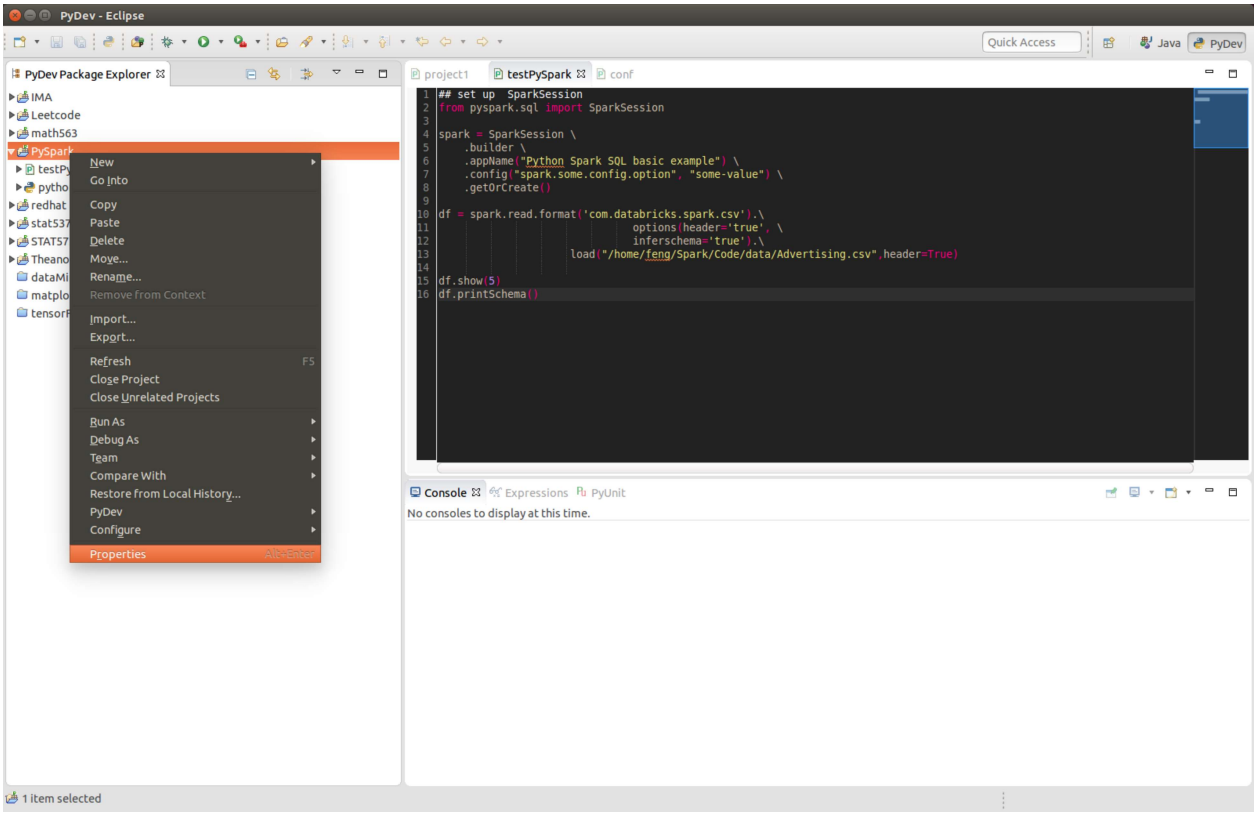
Then you should get the output results in your terminal.



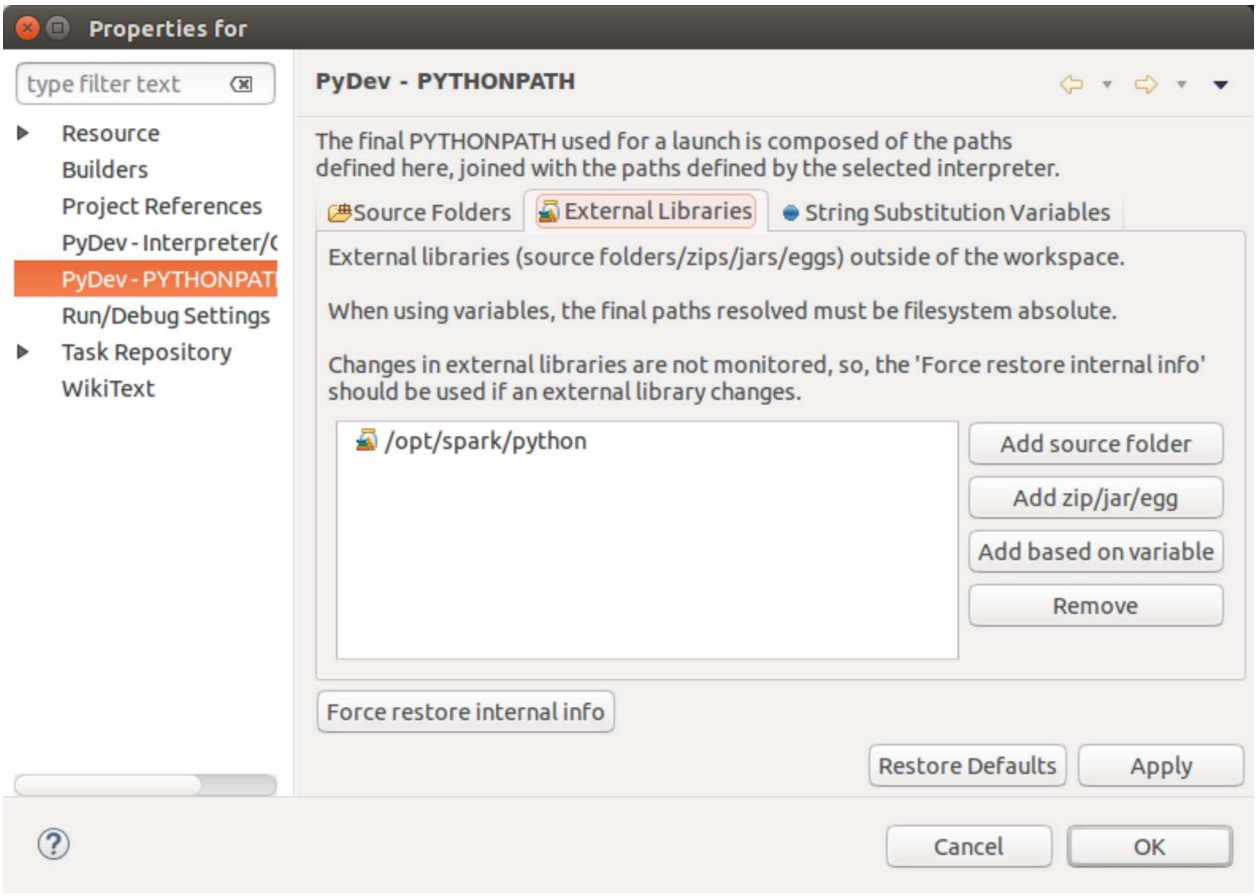
3.4.5 PySpark With Eclipse

If you want to run PySpark code on Eclipse, you need to add the paths for the **External Libraries** for your **Current Project** as follows:

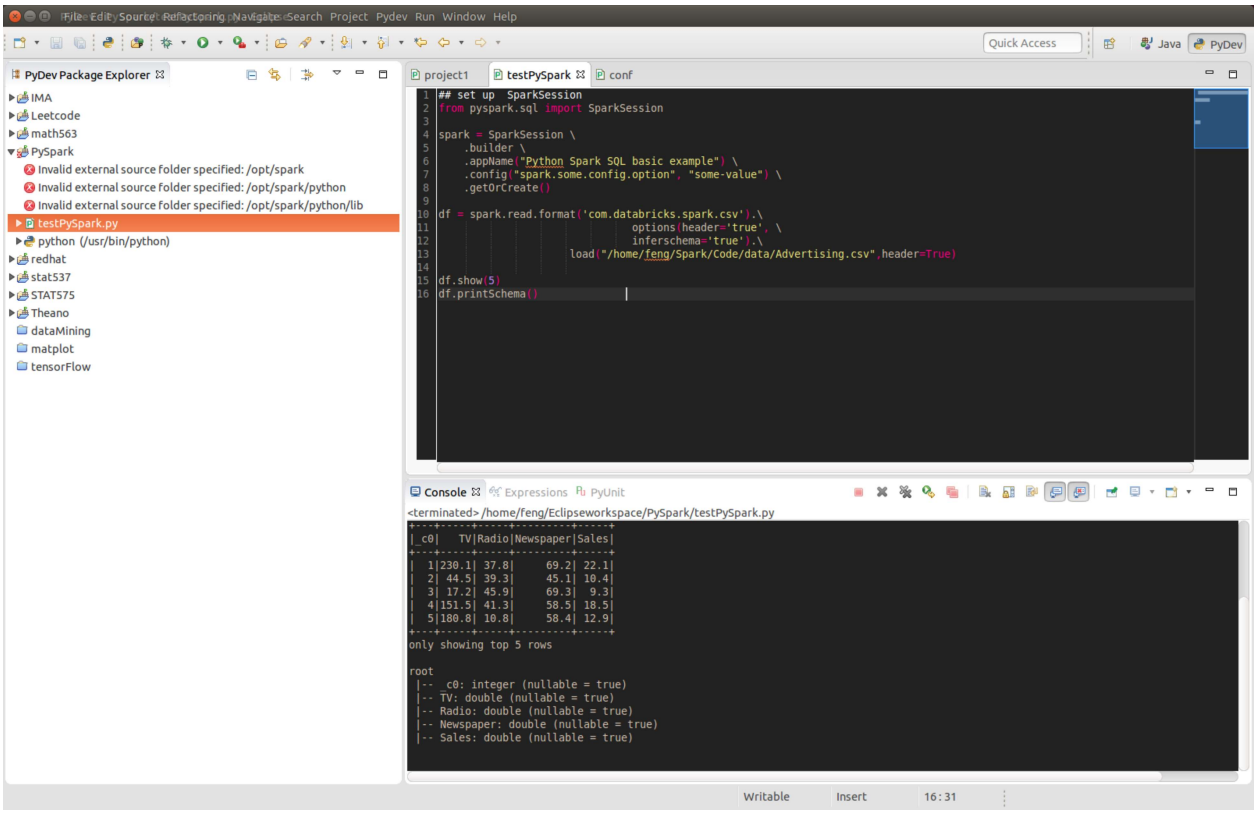
1. Open the properties of your project



2. Add the paths for the **External Libraries**



And then you should be good to run your code on Eclipse with PyDev.



3.5 PySparkling Water: Spark + H2O

1. Download `Sparkling Water` from: https://s3.amazonaws.com/h2o-release/sparkling-water/rel-2.4/5/index.html
2. Test PySparking

```
unzip sparkling-water-2.4.5.zip
cd  ~/sparkling-water-2.4.5/bin
./pysparkling
```

If you have a correct setup for PySpark, then you will get the following results:

```
Using Spark defined in the SPARK_HOME=/Users/dt216661/spark environmental␣
↪property

Python 3.7.1 (default, Dec 14 2018, 13:28:58)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
2019-02-15 14:08:30 WARN  NativeCodeLoader:62 - Unable to load native-hadoop␣
↪library for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
```

(continues on next page)

(continued from previous page)

```
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.
↪properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use␣
↪setLogLevel(newLevel).
2019-02-15 14:08:31 WARN  Utils:66 - Service 'SparkUI' could not bind on port␣
↪4040. Attempting port 4041.
2019-02-15 14:08:31 WARN  Utils:66 - Service 'SparkUI' could not bind on port␣
↪4041. Attempting port 4042.
17/08/30 13:30:12 WARN NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
17/08/30 13:30:17 WARN ObjectStore: Failed to get database global_temp,
returning NoSuchObjectException
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.4.0
      /_/

Using Python version 3.7.1 (default, Dec 14 2018 13:28:58)
SparkSession available as 'spark'.
```

3. Setup `pysparkling` with Jupyter notebook

Add the following alias to your `bashrc` (Linux systems) or `bash_profile` (Mac system)

```
alias sparkling="PYSPARK_DRIVER_PYTHON="ipython" PYSPARK_DRIVER_PYTHON_OPTS= ␣
↪  "notebook" /~/~/sparkling-water-2.4.5/bin/pysparkling"
```

4. Open `pysparkling` in terminal

```
sparkling
```

3.6 Set up Spark on Cloud

Following the setup steps in *Configure Spark on Mac and Ubuntu*, you can set up your own cluster on the cloud, for example AWS, Google Cloud. Actually, for those clouds, they have their own Big Data tool. You can run them directly whitout any setting just like Databricks Community Cloud. If you want more details, please feel free to contact with me.

3.7 PySpark on Colaboratory

Colaboratory is a free Jupyter notebook environment that requires no setup and runs entirely in the cloud.

3.7.1 Installation

```
!pip install pyspark
```

3.7.2 Testing

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

df = spark.sparkContext\
          .parallelize([(1, 2, 3, 'a b c'),
                        (4, 5, 6, 'd e f'),
                        (7, 8, 9, 'g h i')])\
          .toDF(['col1', 'col2', 'col3','col4'])

df.show()
```

Output:

```
+----+----+----+-----+
|col1|col2|col3| col4|
+----+----+----+-----+
|   1|   2|   3|a b c|
|   4|   5|   6|d e f|
|   7|   8|   9|g h i|
+----+----+----+-----+
```

3.8 Demo Code in this Section

The Jupyter notebook can be download from installation on colab.

- Python Source code

```
## set up  SparkSession
from pyspark.sql import SparkSession

spark = SparkSession \
```

(continues on next page)

(continued from previous page)

```
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

df = spark.read.format('com.databricks.spark.csv').\
                                options(header='true', \
                                inferschema='true').\
                    load("/home/feng/Spark/Code/data/Advertising.csv
↪",header=True)

df.show(5)
df.printSchema()
```

CHAPTER

FOUR

AN INTRODUCTION TO APACHE SPARK

Chinese proverb

Know yourself and know your enemy, and you will never be defeated – idiom, from Sunzi's Art of War

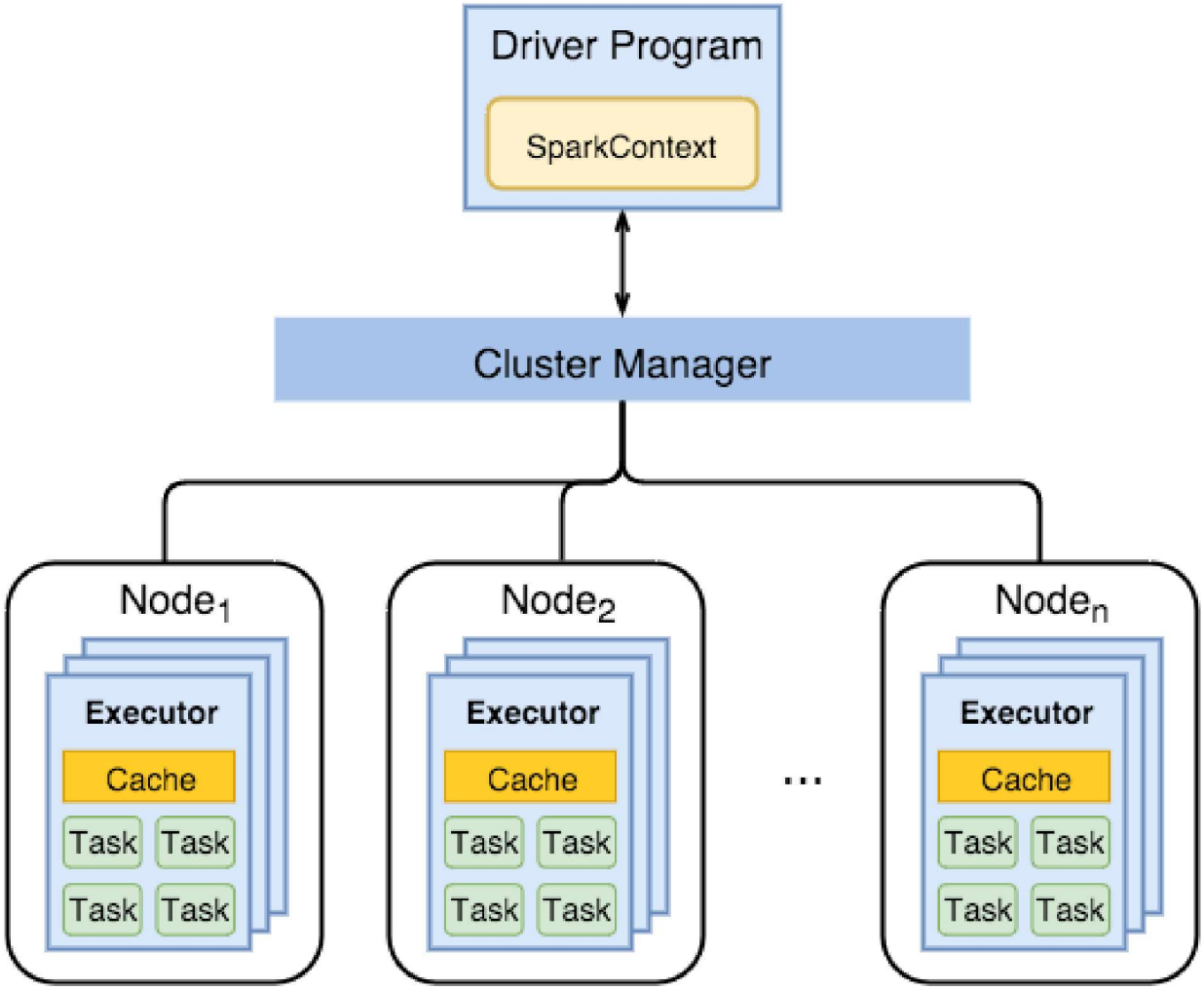
4.1 Core Concepts

Most of the following content comes from [Kirillov2016]. So the copyright belongs to **Anton Kirillov**. I will refer you to get more details from Apache Spark core concepts, architecture and internals.

Before diving deep into how Apache Spark works, lets understand the jargon of Apache Spark

- Job: A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.
- Stages: Jobs are divided into stages. Stages are classified as a Map or reduce stages (Its easier to understand if you have worked on Hadoop and want to correlate). Stages are divided based on computational boundaries, all computations (operators) cannot be Updated in a single Stage. It happens over many stages.
- Tasks: Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor (machine).
- DAG: DAG stands for Directed Acyclic Graph, in the present context its a DAG of operators.
- Executor: The process responsible for executing a task.
- Master: The machine on which the Driver program runs
- Slave: The machine on which the Executor program runs

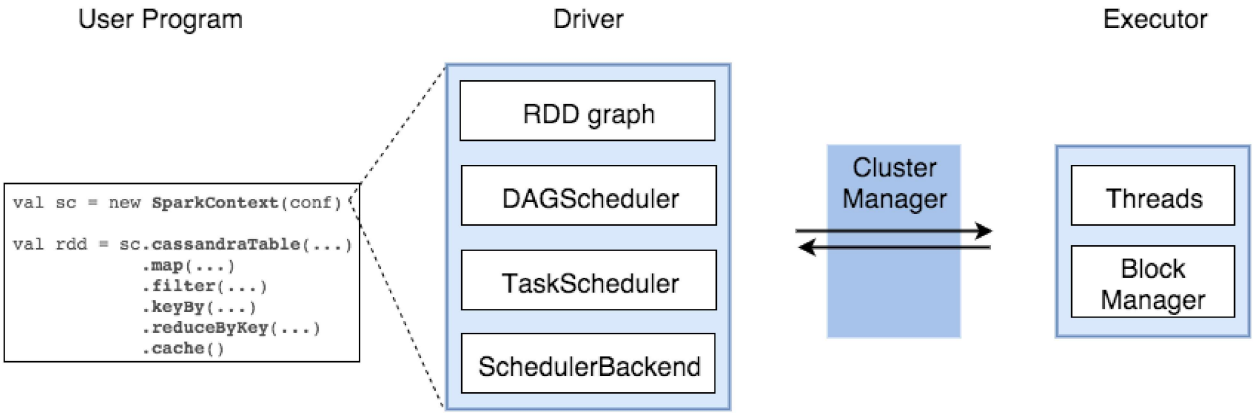
4.2 Spark Components



1. Spark Driver
   - separate process to execute user applications
   - creates SparkContext to schedule jobs execution and negotiate with cluster manager
2. Executors
   - run tasks scheduled by driver
   - store computation results in memory, on disk or off-heap
   - interact with storage systems
3. Cluster Manager
   - Mesos

- YARN
- Spark Standalone

Spark Driver contains more components responsible for translation of user code into actual jobs executed on cluster:



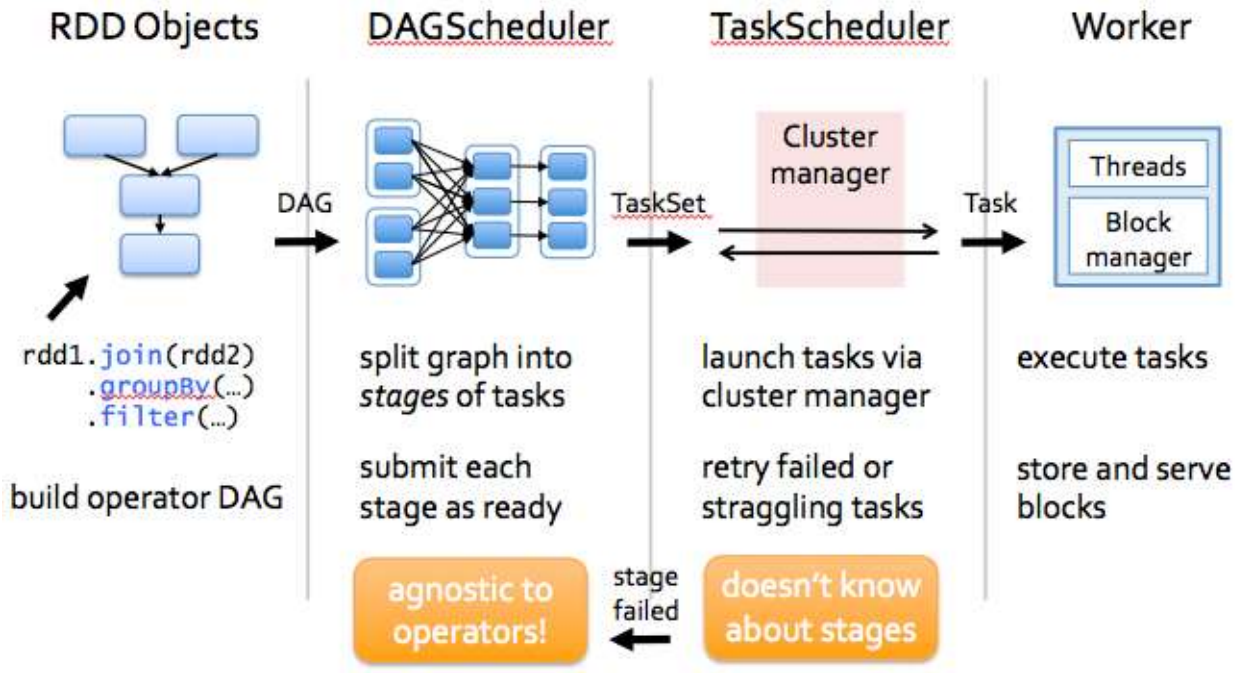
- SparkContext
  - represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster
- DAGScheduler
  - computes a DAG of stages for each job and submits them to TaskScheduler determines preferred locations for tasks (based on cache status or shuffle files locations) and finds minimum schedule to run the jobs
- TaskScheduler
  - responsible for sending tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers
- SchedulerBackend
  - backend interface for scheduling systems that allows plugging in different implementations(Mesos, YARN, Standalone, local)
- BlockManager
  - provides interfaces for putting and retrieving blocks both locally and remotely into various stores (memory, disk, and off-heap)

4.3 Architecture

4.4 How Spark Works?

Spark has a small code base and the system is divided in various layers. Each layer has some responsibilities. The layers are independent of each other.

The first layer is the interpreter, Spark uses a Scala interpreter, with some modifications. As you enter your code in spark console (creating RDD's and applying operators), Spark creates a operator graph. When the user runs an action (like collect), the Graph is submitted to a DAG Scheduler. The DAG scheduler divides operator graph into (map and reduce) stages. A stage is comprised of tasks based on partitions of the input data. The DAG scheduler pipelines operators together to optimize the graph. For e.g. Many map operators can be scheduled in a single stage. This optimization is key to Sparks performance. The final result of a DAG scheduler is a set of stages. The stages are passed on to the Task Scheduler. The task scheduler launches tasks via cluster manager. (Spark Standalone/Yarn/Mesos). The task scheduler doesn't know about dependencies among stages.



CHAPTER

FIVE

PROGRAMMING WITH RDDS

Chinese proverb

If you only know yourself, but not your opponent, you may win or may lose. If you know neither yourself nor your enemy, you will always endanger yourself – idiom, from Sunzi's Art of War

RDD represents **Resilient Distributed Dataset**. An RDD in Spark is simply an immutable distributed collection of objects sets. Each RDD is split into multiple partitions (similar pattern with smaller sets), which may be computed on different nodes of the cluster.

5.1 Create RDD

Usually, there are two popular ways to create the RDDs: loading an external dataset, or distributing a set of collection of objects. The following examples show some simplest ways to create RDDs by using `parallelize()` fucntion which takes an already existing collection in your program and pass the same to the Spark Context.

1. By using `parallelize( )` function

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

df = spark.sparkContext.parallelize([(1, 2, 3, 'a b c'),
              (4, 5, 6, 'd e f'),
              (7, 8, 9, 'g h i')]).toDF(['col1', 'col2', 'col3','col4'])
```

Then you will get the RDD data:

```
df.show()

+----+----+----+-----+
```

(continues on next page)

(continued from previous page)

```
|col1|col2|col3| col4|
+----+----+----+-----+
|   1|   2|   3|a b c|
|   4|   5|   6|d e f|
|   7|   8|   9|g h i|
+----+----+----+-----+
```

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

myData = spark.sparkContext.parallelize([(1,2), (3,4), (5,6), (7,8), (9,10)])
```

Then you will get the RDD data:

```
myData.collect()

[(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
```

2. By using `createDataFrame( )` function

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

Employee = spark.createDataFrame([
                        ('1', 'Joe',   '70000', '1'),
                        ('2', 'Henry', '80000', '2'),
                        ('3', 'Sam',   '60000', '2'),
                        ('4', 'Max',   '90000', '1')],
                        ['Id', 'Name', 'Sallary','DepartmentId']
                       )
```

Then you will get the RDD data:

```
+---+-----+-------+------------+
| Id| Name|Sallary|DepartmentId|
+---+-----+-------+------------+
|  1|  Joe|  70000|           1|
|  2|Henry|  80000|           2|
|  3|  Sam|  60000|           2|
|  4|  Max|  90000|           1|
+---+-----+-------+------------+
```

3. By using `read` and `load` functions

   a. **Read dataset from .csv file**

```
## set up  SparkSession
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

df = spark.read.format('com.databricks.spark.csv').\
                                options(header='true', \
                                inferschema='true').\
                load("/home/feng/Spark/Code/data/Advertising.csv",
↪header=True)

df.show(5)
df.printSchema()
```

   Then you will get the RDD data:

```
+---+-----+-----+---------+-----+
|_c0|   TV|Radio|Newspaper|Sales|
+---+-----+-----+---------+-----+
|  1|230.1| 37.8|     69.2| 22.1|
|  2| 44.5| 39.3|     45.1| 10.4|
|  3| 17.2| 45.9|     69.3|  9.3|
|  4|151.5| 41.3|     58.5| 18.5|
|  5|180.8| 10.8|     58.4| 12.9|
+---+-----+-----+---------+-----+
only showing top 5 rows

root
 |-- _c0: integer (nullable = true)
 |-- TV: double (nullable = true)
 |-- Radio: double (nullable = true)
 |-- Newspaper: double (nullable = true)
 |-- Sales: double (nullable = true)
```

Once created, RDDs offer two types of operations: transformations and actions.

   b. **Read dataset from DataBase**

```
## set up  SparkSession
from pyspark.sql import SparkSession

spark = SparkSession \
            .builder \
            .appName("Python Spark create RDD example") \
            .config("spark.some.config.option", "some-value") \
```

(continues on next page)

(continued from previous page)

```
            .getOrCreate()

## User information
user = 'your_username'
pw   = 'your_password'

## Database information
table_name = 'table_name'
url = 'jdbc:postgresql://##.###.###.##:5432/dataset?user='+user+'&
↪password='+pw
properties ={'driver': 'org.postgresql.Driver', 'password': pw,'user
↪': user}

df = spark.read.jdbc(url=url, table=table_name,␣
↪properties=properties)

df.show(5)
df.printSchema()
```

Then you will get the RDD data:

```
+---+-----+-----+---------+-----+
|_c0|   TV|Radio|Newspaper|Sales|
+---+-----+-----+---------+-----+
|  1|230.1| 37.8|     69.2| 22.1|
|  2| 44.5| 39.3|     45.1| 10.4|
|  3| 17.2| 45.9|     69.3|  9.3|
|  4|151.5| 41.3|     58.5| 18.5|
|  5|180.8| 10.8|     58.4| 12.9|
+---+-----+-----+---------+-----+
only showing top 5 rows

root
 |-- _c0: integer (nullable = true)
 |-- TV: double (nullable = true)
 |-- Radio: double (nullable = true)
 |-- Newspaper: double (nullable = true)
 |-- Sales: double (nullable = true)
```

Note: Reading tables from Database needs the proper drive for the corresponding Database. For example, the above demo needs `org.postgresql.Driver` and you need to download it and put it in `jars` folder of your spark installation path. I download `postgresql-42.1.1.jar` from the official website and put it in `jars` folder.

C. **Read dataset from HDFS**

```
from pyspark.conf import SparkConf
from pyspark.context import SparkContext
from pyspark.sql import HiveContext
```

(continues on next page)

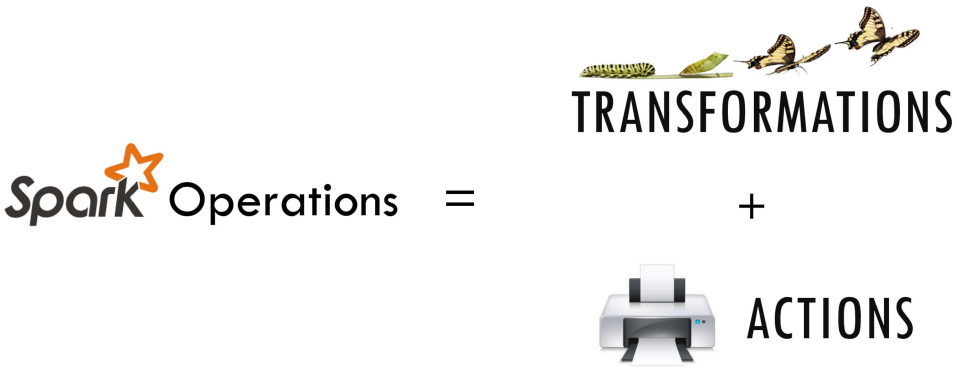
(continued from previous page)

```
sc= SparkContext('local','example')
hc = HiveContext(sc)
tf1 = sc.textFile("hdfs://cdhstltest/user/data/demo.CSV")
print(tf1.first())

hc.sql("use intg_cme_w")
spf = hc.sql("SELECT * FROM spf LIMIT 100")
print(spf.show(5))
```

5.2 Spark Operations

Warning: All the figures below are from Jeffrey Thompson. The interested reader is referred to pyspark pictures

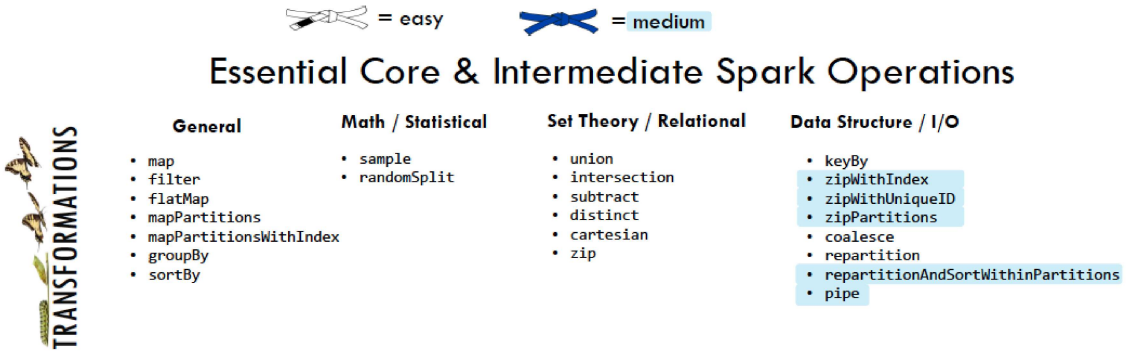
There are two main types of Spark operations: Transformations and Actions [Karau2015].

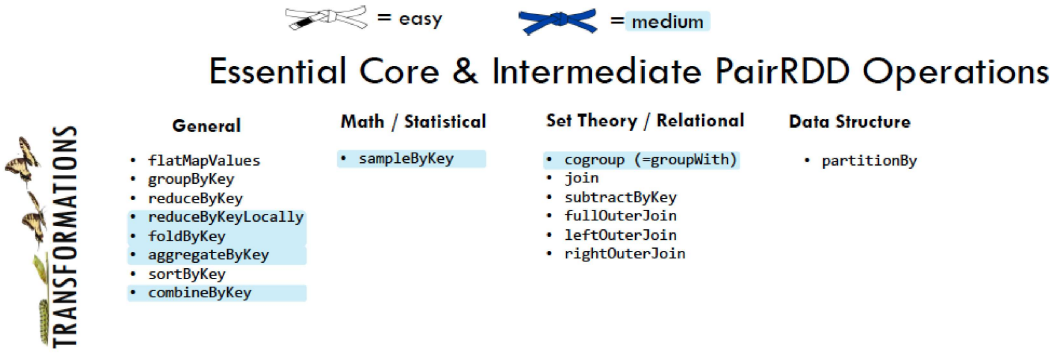


Note: Some people defined three types of operations: Transformations, Actions and Shuffles.

5.2.1 Spark Transformations

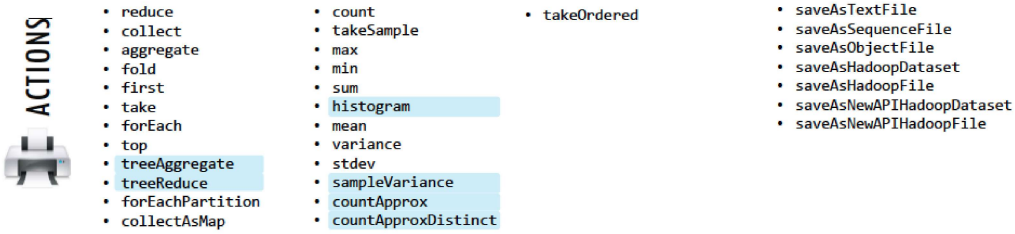
Transformations construct a new RDD from a previous one. For example, one common transformation is filtering data that matches a predicate.

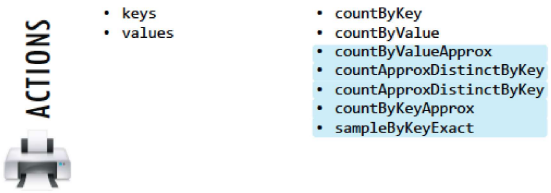




5.2.2 Spark Actions

Actions, on the other hand, compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS).





5.3 rdd.DataFrame vs pd.DataFrame

5.3.1 Create DataFrame

1. From List

```
my_list = [['a', 1, 2], ['b', 2, 3],['c', 3, 4]]
col_name = ['A', 'B', 'C']
```

:: Python Code:

```
# caution for the columns=
pd.DataFrame(my_list,columns= col_name)
#
spark.createDataFrame(my_list, col_name).show()
```

:: Comparison:

```
                  +---+---+---+
                  |  A|  B|  C|
   A  B  C        +---+---+---+
0  a  1  2        |  a|  1|  2|
1  b  2  3        |  b|  2|  3|
2  c  3  4        |  c|  3|  4|
                  +---+---+---+
```

Attention: Pay attentation to the parameter `columns=` in `pd.DataFrame`. Since the default value will make the list as rows.

:: Python Code:

```
# caution for the columns=
pd.DataFrame(my_list, columns= col_name)
#
pd.DataFrame(my_list, col_name)
```

:: Comparison:

```
   A  B  C              0  1  2
0  a  1  2          A   a  1  2
1  b  2  3          B   b  2  3
2  c  3  4          C   c  3  4
```

2. From Dict

```
d = {'A': [0, 1, 0],
     'B': [1, 0, 1],
     'C': [1, 0, 0]}
```

:: Python Code:

```
pd.DataFrame(d)for
# Tedious for PySpark
spark.createDataFrame(np.array(list(d.values())).T.tolist(),list(d.keys())).
↪show()
```

:: Comparison:

```
                    +---+---+---+
                    |  A|  B|  C|
   A  B  C          +---+---+---+
0  0  1  1          |  0|  1|  1|
1  1  0  0          |  1|  0|  0|
2  0  1  0          |  0|  1|  0|
                    +---+---+---+
```

5.3.2 Load DataFrame

1. From DataBase

Most of time, you need to share your code with your colleagues or release your code for Code Review or Quality assurance(QA). You will definitely do not want to have your `User Information` in the code. So you can save them in login.txt:

```
runawayhorse001
PythonTips
```

and use the following code to import your `User Information`:

```
#User Information
try:
    login = pd.read_csv(r'login.txt', header=None)
    user = login[0][0]
    pw = login[0][1]
    print('User information is ready!')
except:
    print('Login information is not available!!!')
```

(continues on next page)

(continued from previous page)

```
#Database information
host = '##.###.###.##'
db_name = 'db_name'
table_name = 'table_name'
```

:: Comparison:

```
conn = psycopg2.connect(host=host, database=db_name, user=user, password=pw)
cur = conn.cursor()

sql = """
      select *
      from {table_name}
      """.format(table_name=table_name)
dp = pd.read_sql(sql, conn)
```

```
# connect to database
url = 'jdbc:postgresql://'+host+':5432/'+db_name+'?user='+user+'&password='+pw
properties ={'driver': 'org.postgresql.Driver', 'password': pw,'user': user}
ds = spark.read.jdbc(url=url, table=table_name, properties=properties)
```

Attention: Reading tables from Database with PySpark needs the proper drive for the corresponding Database. For example, the above demo needs org.postgresql.Driver and you need to download it and put it in `jars` folder of your spark installation path. I download postgresql-42.1.1.jar from the official website and put it in jars folder.

2. From `.csv`

:: Comparison:

```
# pd.DataFrame dp: DataFrame pandas
dp = pd.read_csv('Advertising.csv')
#rdd.DataFrame. dp: DataFrame spark
ds = spark.read.csv(path='Advertising.csv',
#                   sep=',',
#                   encoding='UTF-8',
#                   comment=None,
                    header=True,
                    inferSchema=True)
```

3. From `.json`

Data from: http://api.luftdaten.info/static/v1/data.json

```
dp = pd.read_json("data/data.json")
ds = spark.read.json('data/data.json')
```

:: Python Code:

```
dp[['id','timestamp']].head(4)
#
ds[['id','timestamp']].show(4)
```

:: Comparison:

```
                                                  +----------+------------------
↪-+
                                                  |        id|          ␣
↪timestamp|
            id  timestamp                         +----------+------------------
↪-+
0   2994551481  2019-02-28 17:23:52               |2994551481|2019-02-28␣
↪17:23:52|
1   2994551482  2019-02-28 17:23:52               |2994551482|2019-02-28␣
↪17:23:52|
2   2994551483  2019-02-28 17:23:52               |2994551483|2019-02-28␣
↪17:23:52|
3   2994551484  2019-02-28 17:23:52               |2994551484|2019-02-28␣
↪17:23:52|
                                                  +----------+------------------
↪-+
                                                  only showing top 4 rows
```

5.3.3 First n Rows

:: Python Code:

```
dp.head(4)
#
ds.show(4)
```

:: Comparison:

```
                                    +-----+-----+---------+-----+
                                    |   TV|Radio|Newspaper|Sales|
      TV  Radio  Newspaper  Sales   +-----+-----+---------+-----+
0  230.1   37.8       69.2   22.1   |230.1| 37.8|     69.2| 22.1|
1   44.5   39.3       45.1   10.4   | 44.5| 39.3|     45.1| 10.4|
2   17.2   45.9       69.3    9.3   | 17.2| 45.9|     69.3|  9.3|
3  151.5   41.3       58.5   18.5   |151.5| 41.3|     58.5| 18.5|
                                    +-----+-----+---------+-----+
                                    only showing top 4 rows
```

5.3.4 Column Names

:: Python Code:

```
dp.columns
#
ds.columns
```

:: Comparison:

```
Index(['TV', 'Radio', 'Newspaper', 'Sales'], dtype='object')
['TV', 'Radio', 'Newspaper', 'Sales']
```

5.3.5 Data types

:: Python Code:

```
dp.dtypes
#
ds.dtypes
```

:: Comparison:

```
TV           float64                    [('TV', 'double'),
Radio        float64                     ('Radio', 'double'),
Newspaper    float64                     ('Newspaper', 'double'),
Sales        float64                     ('Sales', 'double')]
dtype: object
```

5.3.6 Fill Null

```
my_list = [['male', 1, None], ['female', 2, 3],['male', 3, 4]]
dp = pd.DataFrame(my_list,columns=['A', 'B', 'C'])
ds = spark.createDataFrame(my_list, ['A', 'B', 'C'])
#
dp.head()
ds.show()
```

:: Comparison:

```
                                    +------+---+----+
                                    |     A|  B|   C|
        A  B    C                   +------+---+----+
0    male  1  NaN                   |  male|  1|null|
1  female  2  3.0                   |female|  2|   3|
2    male  3  4.0                   |  male|  3|   4|
                                    +------+---+----+
```

:: Python Code:

```
dp.fillna(-99)
#
ds.fillna(-99).show()
```

:: Comparison:

```
                                          +------+---+----+
                                          |     A|  B|   C|
        A  B    C                         +------+---+----+
0    male  1  -99                         |  male|  1| -99|
1  female  2  3.0                         |female|  2|   3|
2    male  3  4.0                         |  male|  3|   4|
                                          +------+---+----+
```

5.3.7 Replace Values

:: Python Code:

```
# caution: you need to chose specific col
dp.A.replace(['male', 'female'],[1, 0], inplace=True)
dp
#caution: Mixed type replacements are not supported
ds.na.replace(['male','female'],['1','0']).show()
```

:: Comparison:

```
                                +---+---+----+
                                |  A|  B|   C|
   A  B    C                    +---+---+----+
0  1  1  NaN                    |  1|  1|null|
1  0  2  3.0                    |  0|  2|   3|
2  1  3  4.0                    |  1|  3|   4|
                                +---+---+----+
```

5.3.8 Rename Columns

1. Rename all columns

:: Python Code:

```
dp.columns = ['a','b','c','d']
dp.head(4)
#
ds.toDF('a','b','c','d').show(4)
```

:: Comparison:

```
                                                +-----+----+----+----+
                                                |    a|   b|   c|   d|
```

(continues on next page)

(continued from previous page)

```
       a     b     c     d                    +-----+----+----+----+
0  230.1  37.8  69.2  22.1                    |230.1|37.8|69.2|22.1|
1   44.5  39.3  45.1  10.4                    | 44.5|39.3|45.1|10.4|
2   17.2  45.9  69.3   9.3                    | 17.2|45.9|69.3| 9.3|
3  151.5  41.3  58.5  18.5                    |151.5|41.3|58.5|18.5|
                                              +-----+----+----+----+
                                              only showing top 4 rows
```

2. Rename one or more columns

```
mapping = {'Newspaper':'C','Sales':'D'}
```

:: Python Code:

```
dp.rename(columns=mapping).head(4)
#
new_names = [mapping.get(col,col) for col in ds.columns]
ds.toDF(*new_names).show(4)
```

:: Comparison:

```
                                          +-----+-----+----+----+
                                          |   TV|Radio|   C|   D|
      TV  Radio     C     D               +-----+-----+----+----+
0  230.1   37.8  69.2  22.1               |230.1| 37.8|69.2|22.1|
1   44.5   39.3  45.1  10.4               | 44.5| 39.3|45.1|10.4|
2   17.2   45.9  69.3   9.3               | 17.2| 45.9|69.3| 9.3|
3  151.5   41.3  58.5  18.5               |151.5| 41.3|58.5|18.5|
                                          +-----+-----+----+----+
                                          only showing top 4 rows
```

Note: You can also use `withColumnRenamed` to rename one column in PySpark.

:: Python Code:

```
ds.withColumnRenamed('Newspaper','Paper').show(4
```

:: Comparison:

```
+-----+-----+-----+-----+
|   TV|Radio|Paper|Sales|
+-----+-----+-----+-----+
|230.1| 37.8| 69.2| 22.1|
| 44.5| 39.3| 45.1| 10.4|
| 17.2| 45.9| 69.3|  9.3|
|151.5| 41.3| 58.5| 18.5|
+-----+-----+-----+-----+
only showing top 4 rows
```

5.3.9 Drop Columns

```
drop_name = ['Newspaper','Sales']
```

:: Python Code:

```
dp.drop(drop_name,axis=1).head(4)
#
ds.drop(*drop_name).show(4)
```

:: Comparison:

```
                                +-----+-----+
                                |   TV|Radio|
      TV  Radio                 +-----+-----+
0  230.1   37.8                 |230.1| 37.8|
1   44.5   39.3                 | 44.5| 39.3|
2   17.2   45.9                 | 17.2| 45.9|
3  151.5   41.3                 |151.5| 41.3|
                                +-----+-----+
                                only showing top 4 rows
```

5.3.10 Filter

```
dp = pd.read_csv('Advertising.csv')
#
ds = spark.read.csv(path='Advertising.csv',
                    header=True,
                    inferSchema=True)
```

:: Python Code:

```
dp[dp.Newspaper<20].head(4)
#
ds[ds.Newspaper<20].show(4)
```

:: Comparison:

```
                                                +-----+-----+---------+-----+
                                                |   TV|Radio|Newspaper|Sales|
       TV  Radio  Newspaper  Sales              +-----+-----+---------+-----+
7   120.2   19.6       11.6   13.2              |120.2| 19.6|     11.6| 13.2|
8     8.6    2.1        1.0    4.8              |  8.6|  2.1|      1.0|  4.8|
11  214.7   24.0        4.0   17.4              |214.7| 24.0|      4.0| 17.4|
13   97.5    7.6        7.2    9.7              | 97.5|  7.6|      7.2|  9.7|
                                                +-----+-----+---------+-----+
                                                only showing top 4 rows
```

:: Python Code:

```
dp[(dp.Newspaper<20)&(dp.TV>100)].head(4)
#
ds[(ds.Newspaper<20)&(ds.TV>100)].show(4)
```

:: Comparison:

```
      TV  Radio  Newspaper  Sales
7   120.2   19.6       11.6   13.2
11  214.7   24.0        4.0   17.4
19  147.3   23.9       19.1   14.6
25  262.9    3.5       19.5   12.0
```

```
+-----+-----+---------+-----+
|   TV|Radio|Newspaper|Sales|
+-----+-----+---------+-----+
|120.2| 19.6|     11.6| 13.2|
|214.7| 24.0|      4.0| 17.4|
|147.3| 23.9|     19.1| 14.6|
|262.9|  3.5|     19.5| 12.0|
+-----+-----+---------+-----+
only showing top 4 rows
```

5.3.11 With New Column

:: Python Code:

```
dp['tv_norm'] = dp.TV/sum(dp.TV)
dp.head(4)
#
ds.withColumn('tv_norm', ds.TV/ds.groupBy().agg(F.sum("TV")).collect()[0][0]).
↪show(4)
```

:: Comparison:

```
      TV  Radio  Newspaper  Sales   tv_norm
0  230.1   37.8       69.2   22.1  0.007824
1   44.5   39.3       45.1   10.4  0.001513
2   17.2   45.9       69.3    9.3  0.000585
3  151.5   41.3       58.5   18.5  0.005152
```

```
+-----+-----+---------+-----+--------------------+
|   TV|Radio|Newspaper|Sales|            tv_norm|
+-----+-----+---------+-----+--------------------+
|230.1| 37.8|     69.2| 22.1|0.007824268493802813|
| 44.5| 39.3|     45.1| 10.4|0.001513167961643...|
| 17.2| 45.9|     69.3|  9.3|5.848649200061207E-4|
|151.5| 41.3|     58.5| 18.5|0.005151571824472517|
+-----+-----+---------+-----+--------------------+
only showing top 4 rows
```

:: Python Code:

```
dp['cond'] = dp.apply(lambda c: 1 if ((c.TV>100)&(c.Radio<40)) else 2 if c.
↪Sales> 10 else 3,axis=1)
#
ds.withColumn('cond',F.when((ds.TV>100)&(ds.Radio<40),1)\
                      .when(ds.Sales>10, 2)\
                      .otherwise(3)).show(4)
```

:: Comparison:

```
                                                    +-----+-----+---------+-----+-
↪---+
                                                    |   ␣
↪TV|Radio|Newspaper|Sales|cond|
      TV  Radio  Newspaper  Sales  cond            +-----+-----+---------+-----+-
↪---+
0  230.1   37.8       69.2   22.1     1            |230.1| 37.8|     69.2| 22.1|␣
↪  1|
1   44.5   39.3       45.1   10.4     2            | 44.5| 39.3|     45.1| 10.4|␣
↪  2|
2   17.2   45.9       69.3    9.3     3            | 17.2| 45.9|     69.3|  9.3|␣
↪  3|
3  151.5   41.3       58.5   18.5     2            |151.5| 41.3|     58.5| 18.5|␣
↪  2|
                                                    +-----+-----+---------+-----+-
↪---+
                                                    only showing top 4 rows
```

:: Python Code:

```
dp['log_tv'] = np.log(dp.TV)
dp.head(4)
#
import pyspark.sql.functions as F
ds.withColumn('log_tv',F.log(ds.TV)).show(4)
```

:: Comparison:

```
                                                    +-----+-----+---------+-----+-
↪-----------------+
                                                    |   TV|Radio|Newspaper|Sales|␣
↪           log_tv|
      TV  Radio  Newspaper  Sales    log_tv        +-----+-----+---------+-----+-
↪-----------------+
0  230.1   37.8       69.2   22.1  5.438514        |230.1| 37.8|     69.2| 22.1|␣
↪ 5.43851399704132|
1   44.5   39.3       45.1   10.4  3.795489        | 44.5| 39.3|     45.1| 10.
↪4|3.7954891891721947|
2   17.2   45.9       69.3    9.3  2.844909        | 17.2| 45.9|     69.3|  9.
↪3|2.8449093838194073|
3  151.5   41.3       58.5   18.5  5.020586        |151.5| 41.3|     58.5| 18.5|␣
↪5.020585624949423|
                                                    +-----+-----+---------+-----+-
↪-----------------+
```

(continues on next page)

(continued from previous page)

```
                                                only showing top 4 rows
```

:: Python Code:

```
dp['tv+10'] = dp.TV.apply(lambda x: x+10)
dp.head(4)
#
ds.withColumn('tv+10', ds.TV+10).show(4)
```

:: Comparison:

```
                                                +-----+-----+---------+-----+-
↪----+
                                                |  ␣
↪TV|Radio|Newspaper|Sales|tv+10|
      TV  Radio  Newspaper  Sales  tv+10        +-----+-----+---------+-----+-
↪----+
0  230.1   37.8       69.2   22.1  240.1        |230.1| 37.8|     69.2| 22.
↪1|240.1|
1   44.5   39.3       45.1   10.4   54.5        | 44.5| 39.3|     45.1| 10.4|␣
↪54.5|
2   17.2   45.9       69.3    9.3   27.2        | 17.2| 45.9|     69.3|  9.3|␣
↪27.2|
3  151.5   41.3       58.5   18.5  161.5        |151.5| 41.3|     58.5| 18.
↪5|161.5|
                                                +-----+-----+---------+-----+-
↪----+
                                                only showing top 4 rows
```

5.3.12 Join

```
leftp = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                      'B': ['B0', 'B1', 'B2', 'B3'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']},
                      index=[0, 1, 2, 3])

rightp = pd.DataFrame({'A': ['A0', 'A1', 'A6', 'A7'],
                       'F': ['B4', 'B5', 'B6', 'B7'],
                       'G': ['C4', 'C5', 'C6', 'C7'],
                       'H': ['D4', 'D5', 'D6', 'D7']},
                       index=[4, 5, 6, 7])

lefts = spark.createDataFrame(leftp)
rights = spark.createDataFrame(rightp)
```

```
    A   B   C   D                   A   F   G   H
0  A0  B0  C0  D0               4  A0  B4  C4  D4
1  A1  B1  C1  D1               5  A1  B5  C5  D5
```

(continues on next page)

(continued from previous page)

```
2  A2  B2  C2  D2                6  A6  B6  C6  D6
3  A3  B3  C3  D3                7  A7  B7  C7  D7
```

1. Left Join

:: Python Code:

```
leftp.merge(rightp,on='A',how='left')
#
lefts.join(rights,on='A',how='left')
     .orderBy('A',ascending=True).show()
```

:: Comparison:

```
                                        +---+---+---+---+----+---
↪-+----+
                                        |  A|  B|  C|  D|   F|  ␣
↪G|   H|
   A   B   C   D    F    G    H        +---+---+---+---+----+---
↪-+----+
0  A0  B0  C0  D0   B4   C4   D4       | A0| B0| C0| D0|  B4| ␣
↪C4|  D4|
1  A1  B1  C1  D1   B5   C5   D5       | A1| B1| C1| D1|  B5| ␣
↪C5|  D5|
2  A2  B2  C2  D2  NaN  NaN  NaN       | A2| B2| C2|␣
↪D2|null|null|null|
3  A3  B3  C3  D3  NaN  NaN  NaN       | A3| B3| C3|␣
↪D3|null|null|null|
                                        +---+---+---+---+----+---
↪-+----+
```

2. Right Join

:: Python Code:

```
leftp.merge(rightp,on='A',how='right')
#
lefts.join(rights,on='A',how='right')
     .orderBy('A',ascending=True).show()
```

:: Comparison:

```
                                        +---+----+----+----+---+-
↪--+---+
                                        |  A|   B|   C|   D|  F|␣
↪ G|  H|
   A    B    C    D   F   G   H        +---+----+----+----+---+-
↪--+---+
0  A0   B0   C0   D0  B4  C4  D4       | A0|  B0|  C0|  D0| B4|␣
↪C4| D4|
1  A1   B1   C1   D1  B5  C5  D5       | A1|  B1|  C1|  D1| B5|␣
↪C5| D5|
```

(continues on next page)

(continued from previous page)

```
2  A6  NaN  NaN  NaN  B6  C6  D6          | A6|null|null|null| B6|␣
↪C6| D6|
3  A7  NaN  NaN  NaN  B7  C7  D7          | A7|null|null|null| B7|␣
↪C7| D7|
                                          +---+----+----+----+---+-
↪--+---+
```

3. Inner Join

:: Python Code:

```
leftp.merge(rightp,on='A',how='inner')
#
lefts.join(rights,on='A',how='inner')
     .orderBy('A',ascending=True).show()
```

:: Comparison:

```
                                +---+---+---+---+---+---+---+
                                |  A|  B|  C|  D|  F|  G|  H|
   A   B   C   D   F   G   H    +---+---+---+---+---+---+---+
0  A0  B0  C0  D0  B4  C4  D4    | A0| B0| C0| D0| B4| C4| D4|
1  A1  B1  C1  D1  B5  C5  D5    | A1| B1| C1| D1| B5| C5| D5|
                                +---+---+---+---+---+---+---+
```

4. Full Join

:: Python Code:

```
leftp.merge(rightp,on='A',how='outer')
#
lefts.join(rights,on='A',how='full')
     .orderBy('A',ascending=True).show()
```

:: Comparison:

```
                                          +---+----+----+----+----
↪+----+----+
                                          |  A|   B|   C|   D|  ␣
↪F|   G|   H|
    A    B    C    D    F    G    H      +---+----+----+----+----
↪+----+----+
0  A0   B0   C0   D0   B4   C4   D4      | A0|  B0|  C0|  D0| ␣
↪B4|  C4|  D4|
1  A1   B1   C1   D1   B5   C5   D5      | A1|  B1|  C1|  D1| ␣
↪B5|  C5|  D5|
2  A2   B2   C2   D2  NaN  NaN  NaN      | A2|  B2|  C2| ␣
↪D2|null|null|null|
3  A3   B3   C3   D3  NaN  NaN  NaN      | A3|  B3|  C3| ␣
↪D3|null|null|null|
4  A6  NaN  NaN  NaN   B6   C6   D6      | A6|null|null|null| ␣
↪B6|  C6|  D6|
```

(continues on next page)

(continued from previous page)

```
5  A7   NaN   NaN   NaN    B7    C7    D7      | A7|null|null|null| ␣
↪B7|  C7|  D7|
                                              +---+----+----+----+----
↪+----+----+
```

5.3.13 Concat Columns

```
my_list = [('a', 2, 3),
           ('b', 5, 6),
           ('c', 8, 9),
           ('a', 2, 3),
           ('b', 5, 6),
           ('c', 8, 9)]
col_name = ['col1', 'col2', 'col3']
#
dp = pd.DataFrame(my_list,columns=col_name)
ds = spark.createDataFrame(my_list,schema=col_name)
```

```
  col1  col2  col3
0    a     2     3
1    b     5     6
2    c     8     9
3    a     2     3
4    b     5     6
5    c     8     9
```

:: Python Code:

```
dp['concat'] = dp.apply(lambda x:'%s%s'%(x['col1'],x['col2']),axis=1)
dp
#
ds.withColumn('concat',F.concat('col1','col2')).show()
```

:: Comparison:

```
                                    +----+----+----+------+
                                    |col1|col2|col3|concat|
  col1  col2  col3 concat           +----+----+----+------+
0    a     2     3     a2           |   a|   2|   3|    a2|
1    b     5     6     b5           |   b|   5|   6|    b5|
2    c     8     9     c8           |   c|   8|   9|    c8|
3    a     2     3     a2           |   a|   2|   3|    a2|
4    b     5     6     b5           |   b|   5|   6|    b5|
5    c     8     9     c8           |   c|   8|   9|    c8|
                                    +----+----+----+------+
```

5.3.14 GroupBy

:: Python Code:

```
dp.groupby(['col1']).agg({'col2':'min','col3':'mean'})
#
ds.groupBy(['col1']).agg({'col2': 'min', 'col3': 'avg'}).show()
```

:: Comparison:

```
      col2  col3                          +----+---------+---------+
col1                                      |col1|min(col2)|avg(col3)|
a        2     3                          +----+---------+---------+
b        5     6                          |   c|        8|      9.0|
c        8     9                          |   b|        5|      6.0|
                                          |   a|        2|      3.0|
                                          +----+---------+---------+
```

5.3.15 Pivot

:: Python Code:

```
pd.pivot_table(dp, values='col3', index='col1', columns='col2', aggfunc=np.
↪sum)
#
ds.groupBy(['col1']).pivot('col2').sum('col3').show()
```

:: Comparison:

```
                                  +----+----+----+----+
col2    2     5     8              |col1|   2|   5|   8|
col1                              +----+----+----+----+
a     6.0   NaN   NaN              |   c|null|null|  18|
b     NaN  12.0   NaN              |   b|null|  12|null|
c     NaN   NaN  18.0              |   a|   6|null|null|
                                  +----+----+----+----+
```

5.3.16 Window

```
d = {'A':['a','b','c','d'],'B':['m','m','n','n'],'C':[1,2,3,6]}
dp = pd.DataFrame(d)
ds = spark.createDataFrame(dp)
```

:: Python Code:

```
dp['rank'] = dp.groupby('B')['C'].rank('dense',ascending=False)
#
from pyspark.sql.window import Window
```

(continues on next page)

(continued from previous page)

```
w = Window.partitionBy('B').orderBy(ds.C.desc())
ds = ds.withColumn('rank',F.rank().over(w))
```

:: Comparison:

```
                        +---+---+---+----+
                        |  A|  B|  C|rank|
   A  B  C  rank        +---+---+---+----+
0  a  m  1   2.0        |  b|  m|  2|   1|
1  b  m  2   1.0        |  a|  m|  1|   2|
2  c  n  3   2.0        |  d|  n|  6|   1|
3  d  n  6   1.0        |  c|  n|  3|   2|
                        +---+---+---+----+
```

5.3.17 rank vs dense_rank

```
d ={'Id':[1,2,3,4,5,6],
    'Score': [4.00, 4.00, 3.85, 3.65, 3.65, 3.50]}
#
data = pd.DataFrame(d)
dp = data.copy()
ds = spark.createDataFrame(data)
```

```
   Id  Score
0   1   4.00
1   2   4.00
2   3   3.85
3   4   3.65
4   5   3.65
5   6   3.50
```

:: Python Code:

```
dp['Rank_dense'] = dp['Score'].rank(method='dense',ascending =False)
dp['Rank'] = dp['Score'].rank(method='min',ascending =False)
dp
#
import pyspark.sql.functions as F
from pyspark.sql.window import Window
w = Window.orderBy(ds.Score.desc())
ds = ds.withColumn('Rank_spark_dense',F.dense_rank().over(w))
ds = ds.withColumn('Rank_spark',F.rank().over(w))
ds.show()
```

:: Comparison:

```
                                +---+-----+---------------+----------+
                                | Id|Score|Rank_spark_dense|Rank_spark|
   Id  Score  Rank_dense  Rank  +---+-----+---------------+----------+
```

(continues on next page)

(continued from previous page)

```
0   1   4.00          1.0   1.0  |  1|  4.0|               1|          1|
1   2   4.00          1.0   1.0  |  2|  4.0|               1|          1|
2   3   3.85          2.0   3.0  |  3| 3.85|               2|          3|
3   4   3.65          3.0   4.0  |  4| 3.65|               3|          4|
4   5   3.65          3.0   4.0  |  5| 3.65|               3|          4|
5   6   3.50          4.0   6.0  |  6|  3.5|               4|          6|
                                  +---+-----+----------------+-----------+
```

CHAPTER

SIX

STATISTICS AND LINEAR ALGEBRA PRELIMINARIES

Chinese proverb

If you only know yourself, but not your opponent, you may win or may lose. If you know neither yourself nor your enemy, you will always endanger yourself – idiom, from Sunzi's Art of War

6.1 Notations

- m : the number of the samples
- n : the number of the features
- y_i : i-th label
- $\hat{y}_i$: i-th predicted label
- $\bar{\boldsymbol{y}} = \frac{1}{m}\sum_{i=1}^{m} y_i$: the mean of $\boldsymbol{y}$.
- $\boldsymbol{y}$: the label vector.
- $\hat{\boldsymbol{y}}$: the predicted label vector.

6.2 Linear Algebra Preliminaries

Since I have documented the Linear Algebra Preliminaries in my Prelim Exam note for Numerical Analysis, the interested reader is referred to [Feng2014] for more details (Figure. *Linear Algebra Preliminaries*).

1 Preliminaries

1.1 Linear Algebra Preliminaries

1.1.1 Common Properties

Properties 1.1. *(Structure of Matrices) Let $A = [A_{ij}]$ be a square or rectangular matrix, A is called*

- *diagonal : if $a_{ij} = 0, \forall i \neq j$,*
- *tridiagonal : if $a_{ij} = 0, \forall |i-j| > 1$,*
- *upper triangular : if $a_{ij} = 0, \forall i > j$,*
- *lower triangular : if $a_{ij} = 0, \forall i < j$,*
- *upper Hessenberg : if $a_{ij} = 0, \forall i > j+1$,*
- *lower Hessenberg : if $a_{ij} = 0, \forall j > i+1$,*
- *block diagonal :$A = diag(A_{11}, A_{22}, \cdots, A_{nn})$,*
- *block diagonal :$A = diag(A_{i,i-1}, A_{ii}, \cdots, A_{i,i+1})$.*

Properties 1.2. *(Type of Matrices) Let $A = [A_{ij}]$ be a square or rectangular matrix, A is called*

- *Hermitian : if $A^* = A$,*
- *skew hermitian : if $A^* = -A$,*
- *symmetric : if $A^T = A$,*
- *skew symmetric : if $A^T = -A$,*
- *normal : if $A^T A = AA^T$, when $A \in \mathbb{R}^{n\times n}$, if $A^*A = AA^*$, when $A \in \mathbb{C}^{n\times n}$,*
- *orthogonal : if $A^T A = I$, when $A \in \mathbb{R}^{n\times n}$, unitary : if $A^*A = I$, when $A \in \mathbb{C}^{n\times n}$.*

Properties 1.3. *(Properties of invertible matrices) Let A be $n \times n$ square matrix. If A is invertible , then*

- *$det(A) \neq 0$,*
- *$nullity(A) = 0$,*
- *$rank(A) = n$,*
- *$\lambda_i \neq 0$, (λ_i eigenvalues),*
- *$Ax = b$ has a unique solution for every $b \in \mathbb{R}^n$*
- *$Ax = 0$ has only trivial solution,*
- *the row vectors are linearly independent ,*
- *the column vectors are linearly independent ,*
- *the row vectors of A form a basis for $\mathbb{R}^n$.*
- *the column vectors of A form a basis for $\mathbb{R}^n$,*
- *the row vectors of A span $\mathbb{R}^n$.*
- *the column vectors of A span $\mathbb{R}^n$.*

Properties 1.4. *(Properties of conjugate transpose) Let A, B be $n \times n$ square matrix and γ be a complex constant, then*

- *$(A^*)^* = A$,*
- *$det(A^*) = det(A)$*
- *$(AB)^* = B^*A^*$,*
- *$tr(A^*) = tr(A)$*
- *$(A+B)^* = A^* + B^*$,*
- *$(\gamma A)^* = \gamma^* A^*$.*

Properties 1.5. *(Properties of similar matrices) If $A \sim B$, then*

- *$det(A) = det(B)$,*
- *$rank(A) = rank(B)$,*
- *$eig(A) = eig(B)$,*
- *if $B \sim C$, then $A \sim C$*
- *$A \sim A$,*
- *$B \sim A$*

Page 5 of 236

Fig. 1: Linear Algebra Preliminaries

6.3 Measurement Formula

6.3.1 Mean absolute error

In statistics, **MAE** (Mean absolute error) is a measure of difference between two continuous variables. The Mean Absolute Error is given by:

$$\text{MAE} = \frac{1}{m}\sum_{i=1}^{m}|\hat{y}_i - y_i|.$$

6.3.2 Mean squared error

In statistics, the **MSE** (Mean Squared Error) of an estimator (of a procedure for estimating an unobserved quantity) measures the average of the squares of the errors or deviations—that is, the difference between the estimator and what is estimated.

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$$

6.3.3 Root Mean squared error

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2}$$

6.3.4 Total sum of squares

In statistical data analysis the **TSS** (Total Sum of Squares) is a quantity that appears as part of a standard way of presenting results of such analyses. It is defined as being the sum, over all observations, of the squared differences of each observation from the overall mean.

$$\text{TSS} = \sum_{i=1}^{m}(y_i - \bar{\boldsymbol{y}})^2$$

6.3.5 Explained Sum of Squares

In statistics, the **ESS** (Explained sum of squares), alternatively known as the model sum of squares or sum of squares due to regression.

The ESS is the sum of the squares of the differences of the predicted values and the mean value of the response variable which is given by:

$$\text{ESS} = \sum_{i=1}^{m}(\hat{y}_i - \bar{\boldsymbol{y}})^2$$

6.3.6 Residual Sum of Squares

In statistics, **RSS** (Residual sum of squares), also known as the sum of squared residuals (SSR) or the sum of squared errors of prediction (SSE), is the sum of the squares of residuals which is given by:

$$\text{RSS} = \sum_{i=1}^{m} (\hat{y}_i - y_i)^2$$

6.3.7 Coefficient of determination R^2

$$R^2 := \frac{ESS}{TSS} = 1 - \frac{\text{RSS}}{\text{TSS}}.$$

Note: In general ($\boldsymbol{y}^T\bar{\boldsymbol{y}} = \hat{\boldsymbol{y}}^T\bar{\boldsymbol{y}}$), total sum of squares = explained sum of squares + residual sum of squares, i.e.:

$$\text{TSS} = \text{ESS} + \text{RSS} \text{ if and only if } \boldsymbol{y}^T\bar{\boldsymbol{y}} = \hat{\boldsymbol{y}}^T\bar{\boldsymbol{y}}.$$

More details can be found at Partitioning in the general ordinary least squares model.

6.4 Confusion Matrix

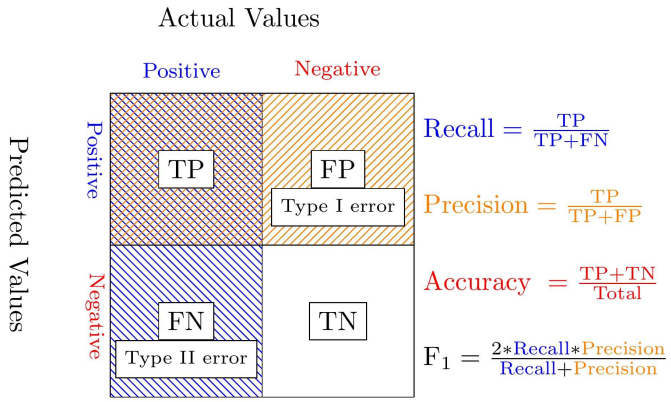


Fig. 2: Confusion Matrix

6.4.1 Recall

$$\text{Recall} = \frac{\text{TP}}{\text{TP+FN}}$$

6.4.2 Precision

$$\text{Precision} = \frac{\text{TP}}{\text{TP+FP}}$$

6.4.3 Accuracy

$$\text{Accuracy } = \frac{\text{TP+TN}}{\text{Total}}$$

6.4.4 F_1-score

$$\text{F}_1 = \frac{2 * \text{Recall} * \text{Precision}}{\text{Recall} + \text{Precision}}$$

6.5 Statistical Tests

6.5.1 Correlational Test

- Pearson correlation: Tests for the strength of the association between two continuous variables.
- Spearman correlation: Tests for the strength of the association between two ordinal variables (does not rely on the assumption of normal distributed data).
- Chi-square: Tests for the strength of the association between two categorical variables.

6.5.2 Comparison of Means test

- Paired T-test: Tests for difference between two related variables.
- Independent T-test: Tests for difference between two independent variables.
- ANOVA: Tests the difference between group means after any other variance in the outcome variable is accounted for.

6.5.3 Non-parametric Test

- Wilcoxon rank-sum test: Tests for difference between two independent variables - takes into account magnitude and direction of difference.
- Wilcoxon sign-rank test: Tests for difference between two related variables - takes into account magnitude and direction of difference.
- Sign test: Tests if two related variables are different – ignores magnitude of change, only takes into account direction.

CHAPTER

SEVEN

DATA EXPLORATION

Chinese proverb

A journey of a thousand miles begins with a single step – idiom, from Laozi.

I wouldn't say that understanding your dataset is the most difficult thing in data science, but it is really important and time-consuming. Data Exploration is about describing the data by means of statistical and visualization techniques. We explore data in order to understand the features and bring important features to our models.

7.1 Univariate Analysis

In mathematics, univariate refers to an expression, equation, function or polynomial of only one variable. "Uni" means "one", so in other words your data has only one variable. So you do not need to deal with the causes or relationships in this step. Univariate analysis takes data, summarizes that variables (attributes) one by one and finds patterns in the data.

There are many ways that can describe patterns found in univariate data include central tendency (mean, mode and median) and dispersion: range, variance, maximum, minimum, quartiles (including the interquartile range), coefficient of variation and standard deviation. You also have several options for visualizing and describing data with univariate data. Such as `frequency Distribution Tables`, `bar Charts`, `histograms`, `frequency Polygons`, `pie Charts`.

The variable could be either categorical or numerical, I will demostrate the different statistical and visulization techniques to investigate each type of the variable.

- The Jupyter notebook can be download from Data Exploration.
- The data can be download from German Credit.

7.1.1 Numerical Variables

Describe

The `describe` function in `pandas` and `spark` will give us most of the statistical results, such as `min`, `median`, `max`, `quartiles` and `standard deviation`. With the help of the user defined function, you can get even more statistical results.

```
# selected varables for the demonstration
num_cols = ['Account Balance','No of dependents']
df.select(num_cols).describe().show()
```

```
+-------+------------------+-------------------+
|summary|   Account Balance|   No of dependents|
+-------+------------------+-------------------+
|  count|              1000|               1000|
|   mean|             2.577|              1.155|
| stddev|1.2576377271108936|0.36208577175319395|
|    min|                 1|                  1|
|    max|                 4|                  2|
+-------+------------------+-------------------+
```

You may find out that the default function in PySpark does not include the quartiles. The following function will help you to get the same results in Pandas

```
def describe_pd(df_in, columns, deciles=False):
    '''
    Function to union the basic stats results and deciles
    :param df_in: the input dataframe
    :param columns: the cloumn name list of the numerical variable
    :param deciles: the deciles output

    :return : the numerical describe info. of the input dataframe

    :author: Ming Chen and Wenqiang Feng
    :email:  von198@gmail.com
    '''

    if deciles:
        percentiles = np.array(range(0, 110, 10))
    else:
        percentiles = [25, 50, 75]

    percs = np.transpose([np.percentile(df_in.select(x).collect(),␣
↪percentiles) for x in columns])
    percs = pd.DataFrame(percs, columns=columns)
    percs['summary'] = [str(p) + '%' for p in percentiles]

    spark_describe = df_in.describe().toPandas()
    new_df = pd.concat([spark_describe, percs],ignore_index=True)
    new_df = new_df.round(2)
    return new_df[['summary'] + columns]
```

```
describe_pd(df,num_cols)
```

```
+-------+------------------+-----------------+
|summary|   Account Balance| No of dependents|
+-------+------------------+-----------------+
|  count|            1000.0|           1000.0|
|   mean|             2.577|            1.155|
| stddev|1.2576377271108936|0.362085771753194|
|    min|               1.0|              1.0|
|    max|               4.0|              2.0|
|    25%|               1.0|              1.0|
|    50%|               2.0|              1.0|
|    75%|               4.0|              1.0|
+-------+------------------+-----------------+
```

Sometimes, because of the confidential data issues, you can not deliver the real data and your clients may ask more statistical results, such as `deciles`. You can apply the follwing function to achieve it.

```
describe_pd(df,num_cols,deciles=True)
```

```
+-------+------------------+-----------------+
|summary|   Account Balance| No of dependents|
+-------+------------------+-----------------+
|  count|            1000.0|           1000.0|
|   mean|             2.577|            1.155|
| stddev|1.2576377271108936|0.362085771753194|
|    min|               1.0|              1.0|
|    max|               4.0|              2.0|
|     0%|               1.0|              1.0|
|    10%|               1.0|              1.0|
|    20%|               1.0|              1.0|
|    30%|               2.0|              1.0|
|    40%|               2.0|              1.0|
|    50%|               2.0|              1.0|
|    60%|               3.0|              1.0|
|    70%|               4.0|              1.0|
|    80%|               4.0|              1.0|
|    90%|               4.0|              2.0|
|   100%|               4.0|              2.0|
+-------+------------------+-----------------+
```

Skewness and Kurtosis

This subsection comes from Wikipedia Skewness.

In probability theory and statistics, skewness is a measure of the asymmetry of the probability distribution of a real-valued random variable about its mean. The skewness value can be positive or negative, or undefined.For a unimodal distribution, negative skew commonly indicates that the tail is on the left side of the distribution, and positive skew indicates that the tail is on the right.

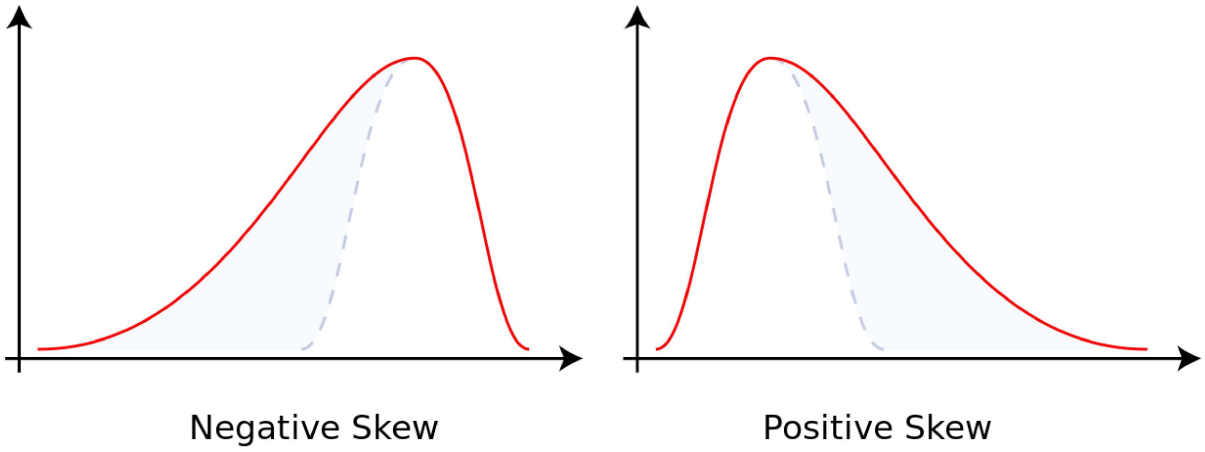
Consider the two distributions in the figure just below. Within each graph, the values on the right side of the

distribution taper differently from the values on the left side. These tapering sides are called tails, and they provide a visual means to determine which of the two kinds of skewness a distribution has:

1. negative skew: The left tail is longer; the mass of the distribution is concentrated on the right of the figure. The distribution is said to be left-skewed, left-tailed, or skewed to the left, despite the fact that the curve itself appears to be skewed or leaning to the right; left instead refers to the left tail being drawn out and, often, the mean being skewed to the left of a typical center of the data. A left-skewed distribution usually appears as a right-leaning curve.
2. positive skew: The right tail is longer; the mass of the distribution is concentrated on the left of the figure. The distribution is said to be right-skewed, right-tailed, or skewed to the right, despite the fact that the curve itself appears to be skewed or leaning to the left; right instead refers to the right tail being drawn out and, often, the mean being skewed to the right of a typical center of the data. A right-skewed distribution usually appears as a left-leaning curve.

This subsection comes from Wikipedia Kurtosis.

In probability theory and statistics, kurtosis (kyrtos or kurtos, meaning "curved, arching") is a measure of the "tailedness" of the probability distribution of a real-valued random variable. In a similar way to the concept of skewness, kurtosis is a descriptor of the shape of a probability distribution and, just as for skewness, there are different ways of quantifying it for a theoretical distribution and corresponding ways of estimating it from a sample from a population.




```
from pyspark.sql.functions import col, skewness, kurtosis
df.select(skewness(var),kurtosis(var)).show()
```

```
+--------------------+--------------------+
|skewness(Age (years))|kurtosis(Age (years))|
+--------------------+--------------------+
|   1.0231743160548064|   0.6114371688367672|
+--------------------+--------------------+
```

Warning: Sometimes the statistics can be misleading!

F. J. Anscombe once said that make both calculations and graphs. Both sorts of output should be studied; each will contribute to understanding. These 13 datasets in Figure *Same Stats, Different Graphs* (the Datasaurus, plus 12 others) each have the same summary statistics (x/y mean, x/y standard deviation, and Pearson's correlation) to two decimal places, while being drastically different in appearance. This work describes the technique we developed to create this dataset, and others like it. More details and interesting results can be found in Same Stats Different Graphs.

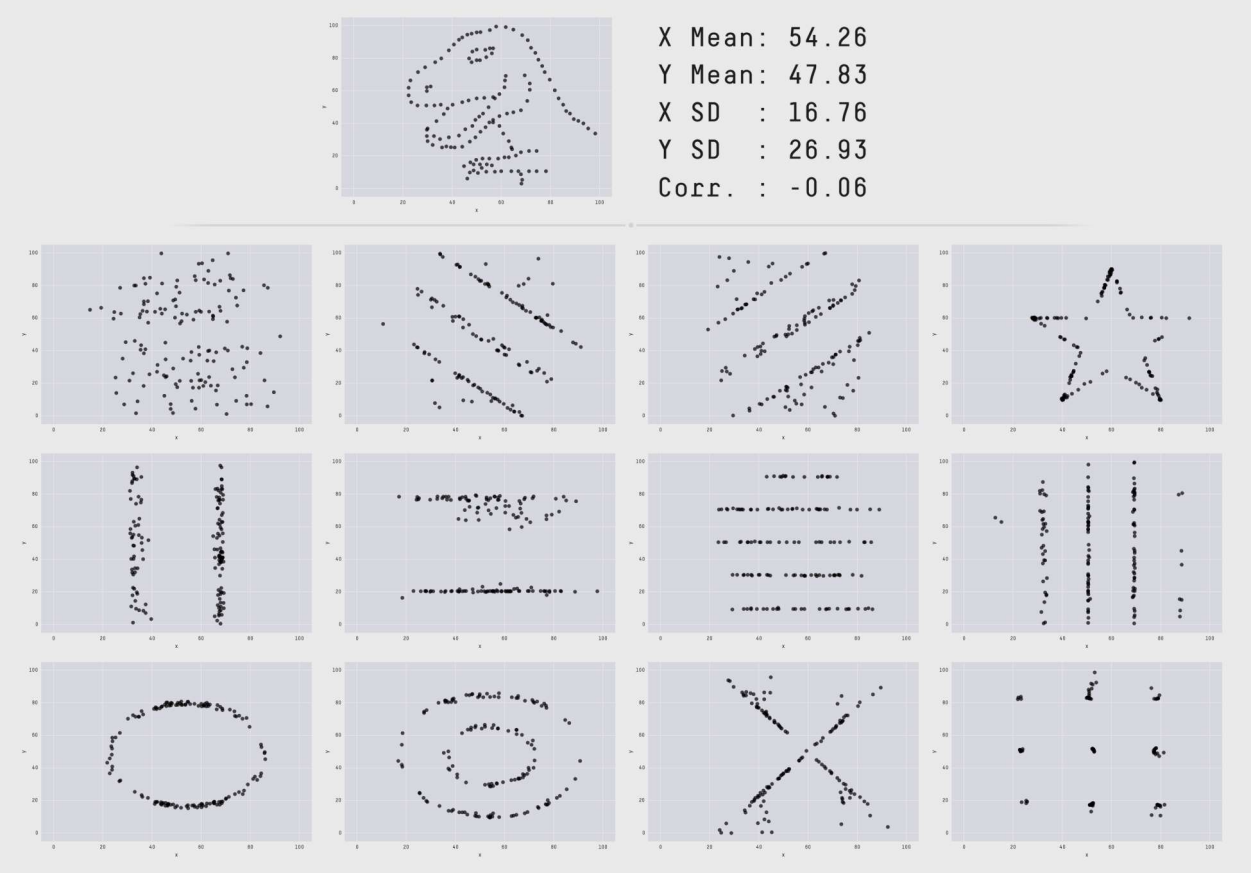


Fig. 1: Same Stats, Different Graphs

Histogram

Warning: Histograms are often confused with Bar graphs!

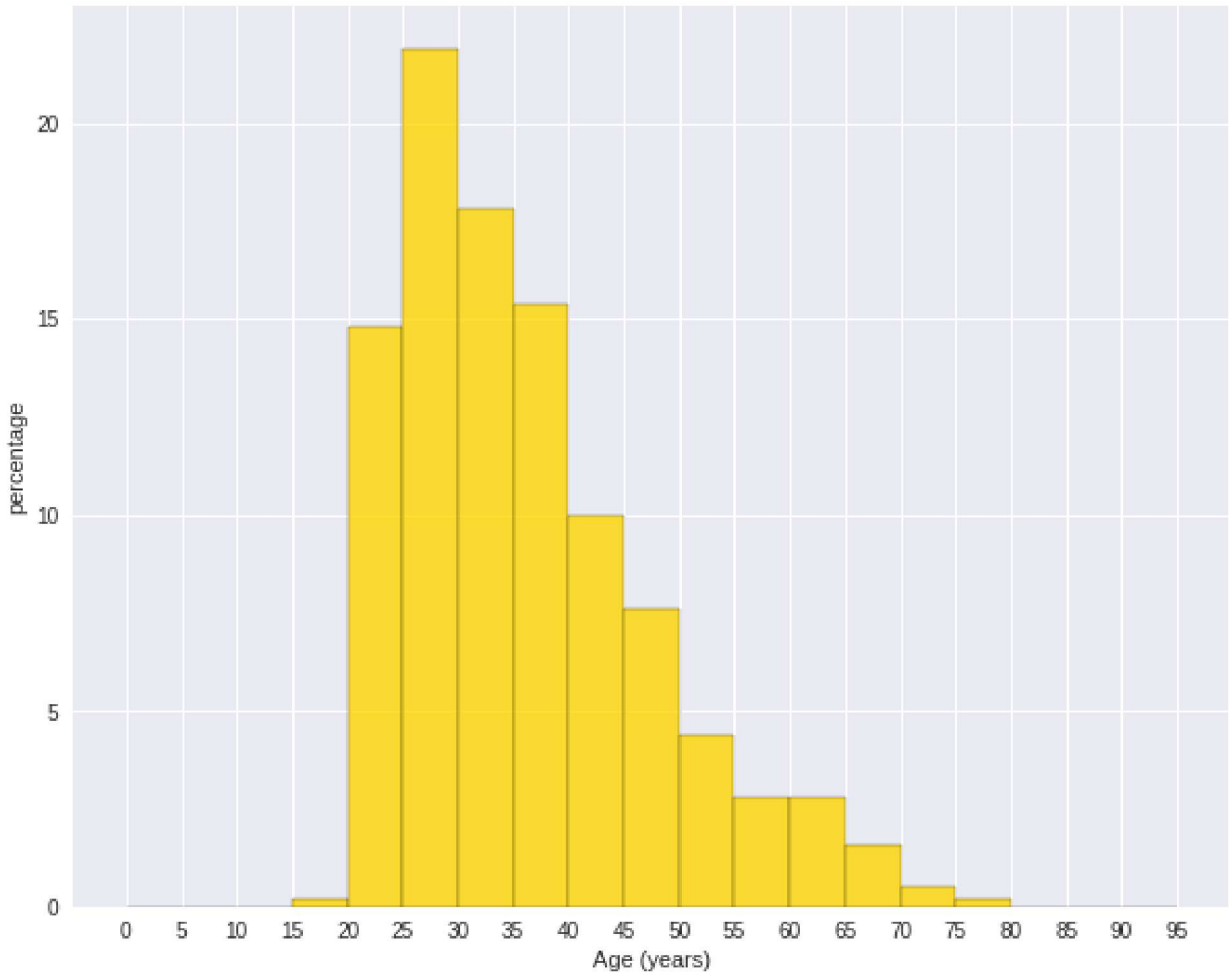
The fundamental difference between histogram and bar graph will help you to identify the two easily is that there are gaps between bars in a bar graph but in the histogram, the bars are adjacent to each other. The interested reader is referred to Difference Between Histogram and Bar Graph.

```
var = 'Age (years)'
x = data1[var]
bins = np.arange(0, 100, 5.0)
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(10,8))
# the histogram of the data
plt.hist(x, bins, alpha=0.8, histtype='bar', color='gold',
         ec='black',weights=np.zeros_like(x) + 100. / x.size)

plt.xlabel(var)
plt.ylabel('percentage')
plt.xticks(bins)
plt.show()

fig.savefig(var+".pdf", bbox_inches='tight')
```

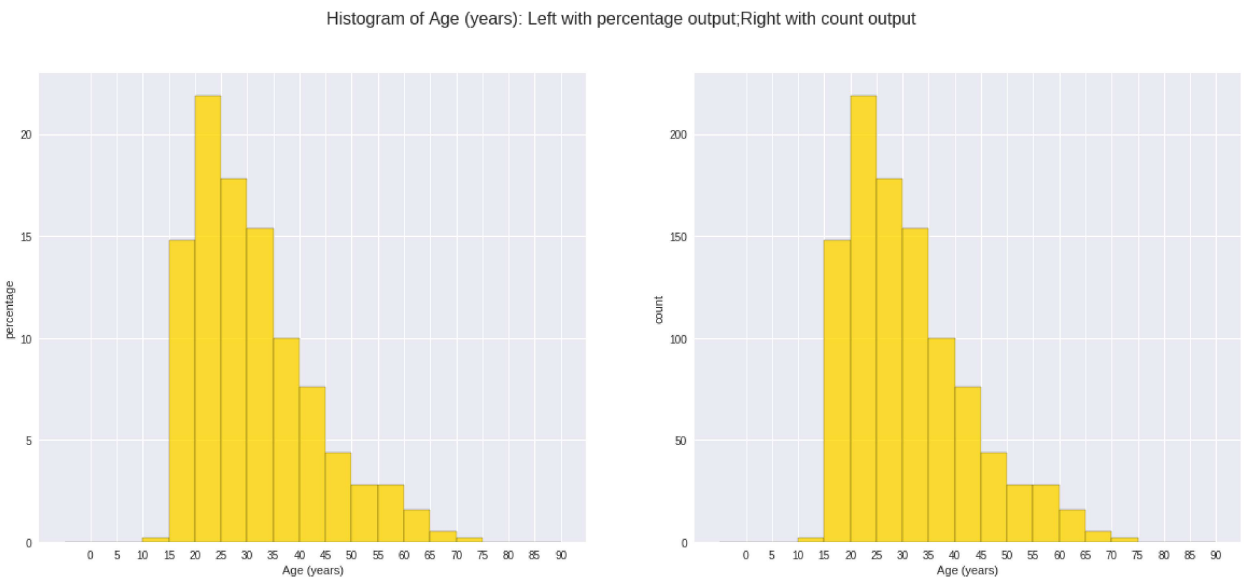



```
var = 'Age (years)'
x = data1[var]
bins = np.arange(0, 100, 5.0)
```

(continues on next page)

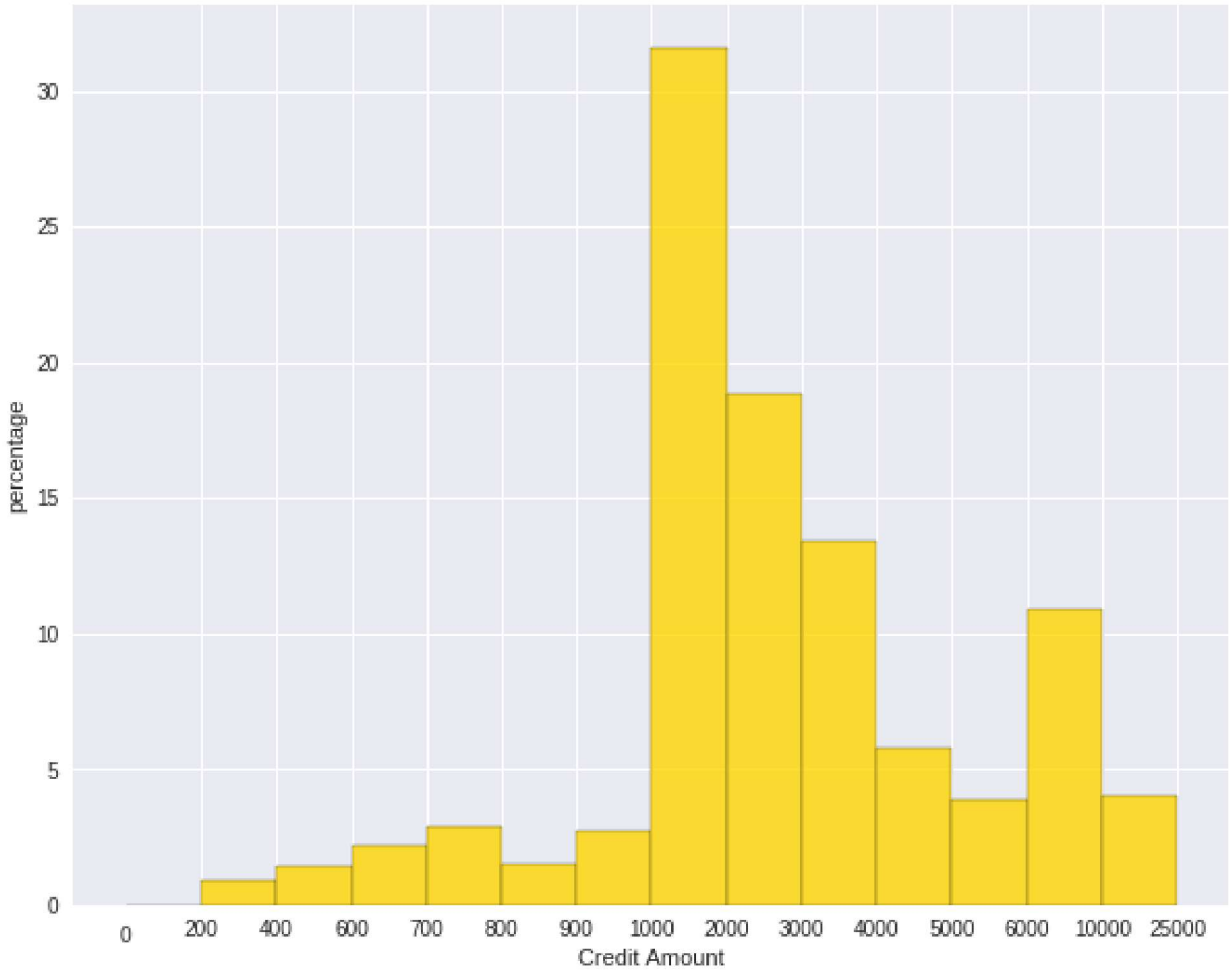
(continued from previous page)

```
#########################################################################
hist, bin_edges = np.histogram(x,bins,
                               weights=np.zeros_like(x) + 100. / x.size)
# make the histogram

fig = plt.figure(figsize=(20, 8))
ax = fig.add_subplot(1, 2, 1)

# Plot the histogram heights against integers on the x axis
ax.bar(range(len(hist)),hist,width=1,alpha=0.8,ec ='black', color='gold')
# # Set the ticks to the middle of the bars
ax.set_xticks([0.5+i for i,j in enumerate(hist)])
# Set the xticklabels to a string that tells us what the bin edges were
labels =['{}'.format(int(bins[i+1])) for i,j in enumerate(hist)]
labels.insert(0,'0')
ax.set_xticklabels(labels)
plt.xlabel(var)
plt.ylabel('percentage')

#########################################################################

hist, bin_edges = np.histogram(x,bins) # make the histogram

ax = fig.add_subplot(1, 2, 2)
# Plot the histogram heights against integers on the x axis
ax.bar(range(len(hist)),hist,width=1,alpha=0.8,ec ='black', color='gold')

# # Set the ticks to the middle of the bars
ax.set_xticks([0.5+i for i,j in enumerate(hist)])

# Set the xticklabels to a string that tells us what the bin edges were
labels =['{}'.format(int(bins[i+1])) for i,j in enumerate(hist)]
labels.insert(0,'0')
ax.set_xticklabels(labels)
plt.xlabel(var)
plt.ylabel('count')
plt.suptitle('Histogram of {}: Left with percentage output;Right with count␣
↪output'
             .format(var), size=16)
plt.show()

fig.savefig(var+".pdf", bbox_inches='tight')
```



Sometimes, some people will ask you to plot the unequal width (invalid argument for histogram) of the bars. You can still achieve it by the following trick.

```
var = 'Credit Amount'
plot_data = df.select(var).toPandas()
x= plot_data[var]

bins =[0,200,400,600,700,800,900,1000,2000,3000,4000,5000,6000,10000,25000]

hist, bin_edges = np.histogram(x,bins,weights=np.zeros_like(x) + 100. / x.
↪size) # make the histogram

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(1, 1, 1)
# Plot the histogram heights against integers on the x axis
ax.bar(range(len(hist)),hist,width=1,alpha=0.8,ec ='black',color = 'gold')

# # Set the ticks to the middle of the bars
ax.set_xticks([0.5+i for i,j in enumerate(hist)])

# Set the xticklabels to a string that tells us what the bin edges were
#labels =['{}k'.format(int(bins[i+1]/1000)) for i,j in enumerate(hist)]
labels =['{}'.format(bins[i+1]) for i,j in enumerate(hist)]
labels.insert(0,'0')
ax.set_xticklabels(labels)
#plt.text(-0.6, -1.4,'0')
plt.xlabel(var)
plt.ylabel('percentage')
plt.show()
```



Box plot and violin plot

Note that although violin plots are closely related to Tukey's (1977) box plots, the violin plot can show more information than box plot. When we perform an exploratory analysis, nothing about the samples could be known. So the distribution of the samples can not be assumed to a normal distribution and usually when you get a big data, the normal distribution will show some out liars in box plot.

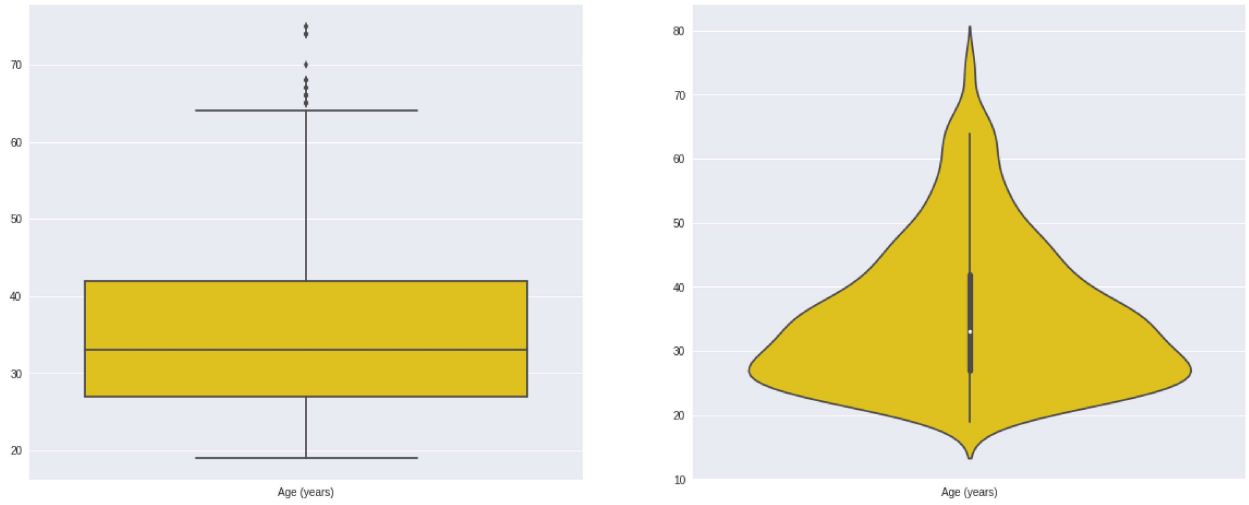
However, the violin plots are potentially misleading for smaller sample sizes, where the density plots can appear to show interesting features (and group-differences therein) even when produced for standard normal data. Some poster suggested the sample size should larger that 250. The sample sizes (e.g. n>250 or ideally even larger), where the kernel density plots provide a reasonably accurate representation of the distributions, potentially showing nuances such as bimodality or other forms of non-normality that would be invisible or less clear in box plots. More details can be found in A simple comparison of box plots and violin plots.

```
x = df.select(var).toPandas()

fig = plt.figure(figsize=(20, 8))
ax = fig.add_subplot(1, 2, 1)
ax = sns.boxplot(data=x)
```

(continues on next page)

(continued from previous page)

```
ax = fig.add_subplot(1, 2, 2)
ax = sns.violinplot(data=x)
```



7.1.2 Categorical Variables

Compared with the numerical variables, the categorical variables are much more easier to do the exploration.

Frequency table

```
from pyspark.sql import functions as F
from pyspark.sql.functions import rank,sum,col
from pyspark.sql import Window

window = Window.rowsBetween(Window.unboundedPreceding,Window.
↪unboundedFollowing)
# withColumn('Percent %',F.format_string("%5.0f%%\n",col('Credit_num')*100/
↪col('total'))).\
tab = df.select(['age_class','Credit Amount']).\
   groupBy('age_class').\
   agg(F.count('Credit Amount').alias('Credit_num'),
       F.mean('Credit Amount').alias('Credit_avg'),
       F.min('Credit Amount').alias('Credit_min'),
       F.max('Credit Amount').alias('Credit_max')).\
   withColumn('total',sum(col('Credit_num')).over(window)).\
   withColumn('Percent',col('Credit_num')*100/col('total')).\
   drop(col('total'))
```

```
+---------+----------+------------------+----------+----------+-------+
|age_class|Credit_num|        Credit_avg|Credit_min|Credit_max|Percent|
+---------+----------+------------------+----------+----------+-------+
```

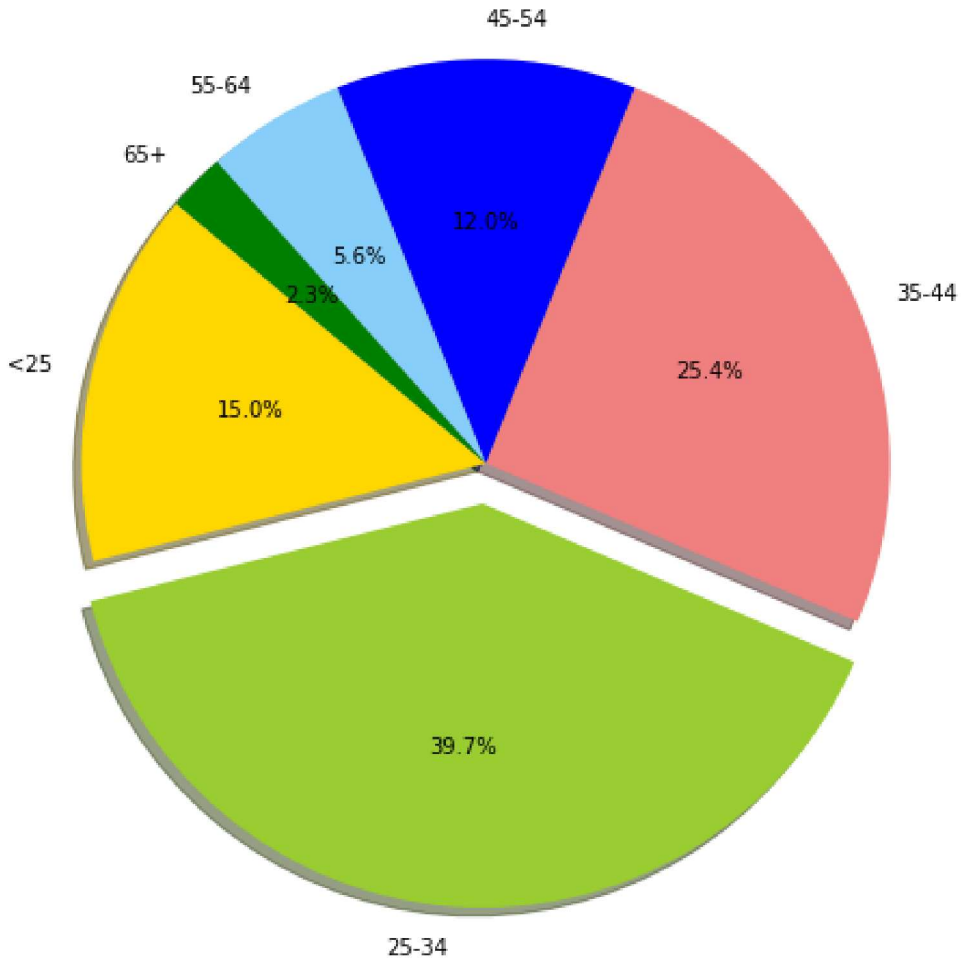
(continues on next page)

(continued from previous page)

```
|    45-54|       120|3183.0666666666666|       338|     12612|   12.0|
|      <25|       150| 2970.733333333333|       276|     15672|   15.0|
|    55-64|        56| 3493.660714285714|       385|     15945|    5.6|
|    35-44|       254| 3403.771653543307|       250|     15857|   25.4|
|    25-34|       397| 3298.823677581864|       343|     18424|   39.7|
|      65+|        23|3210.1739130434785|       571|     14896|    2.3|
+---------+----------+------------------+----------+----------+-------+
```

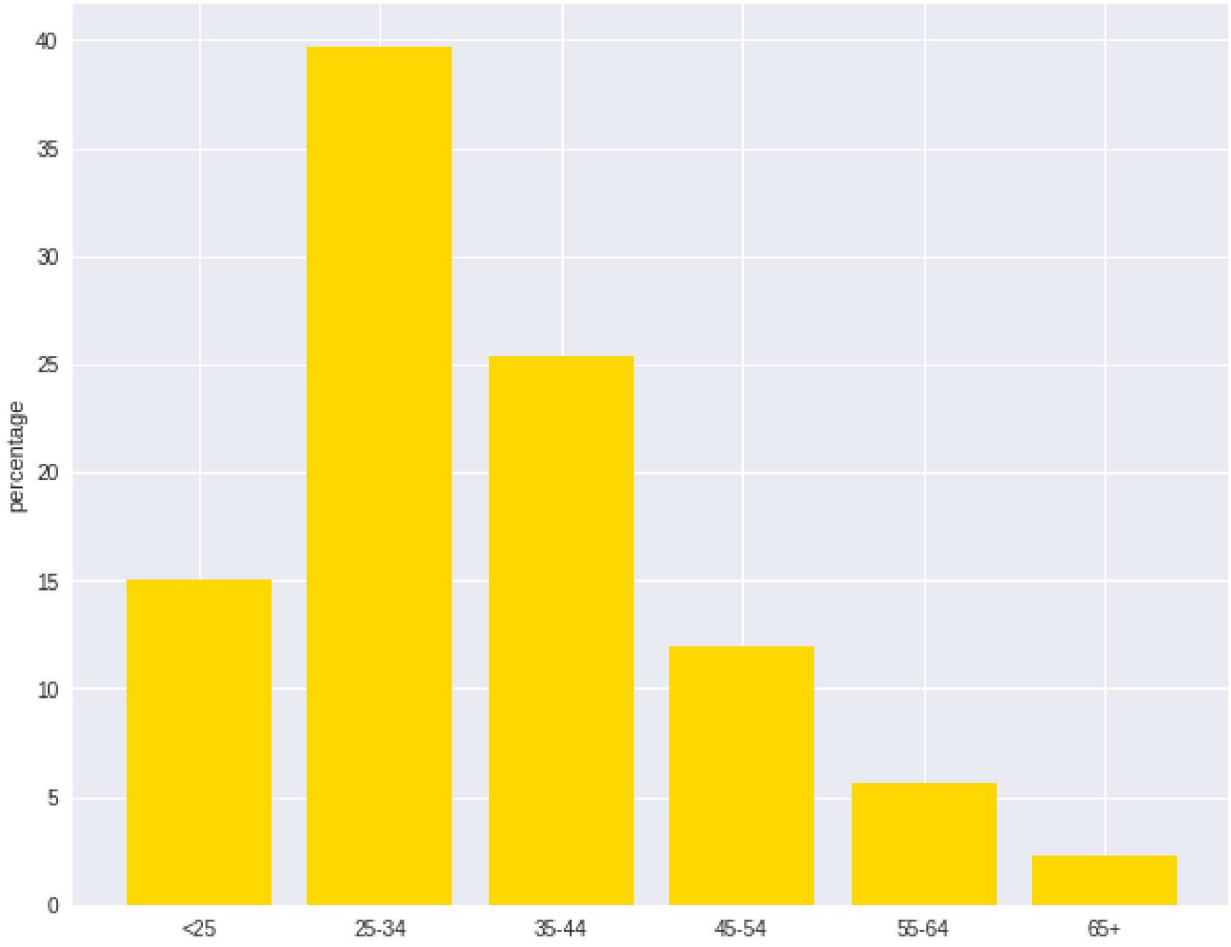
Pie plot

```
# Data to plot
labels = plot_data.age_class
sizes =  plot_data.Percent
colors = ['gold', 'yellowgreen', 'lightcoral','blue', 'lightskyblue','green',
↪'red']
explode = (0, 0.1, 0, 0,0,0)  # explode 1st slice

# Plot
plt.figure(figsize=(10,8))
plt.pie(sizes, explode=explode, labels=labels, colors=colors,
        autopct='%1.1f%%', shadow=True, startangle=140)

plt.axis('equal')
plt.show()
```

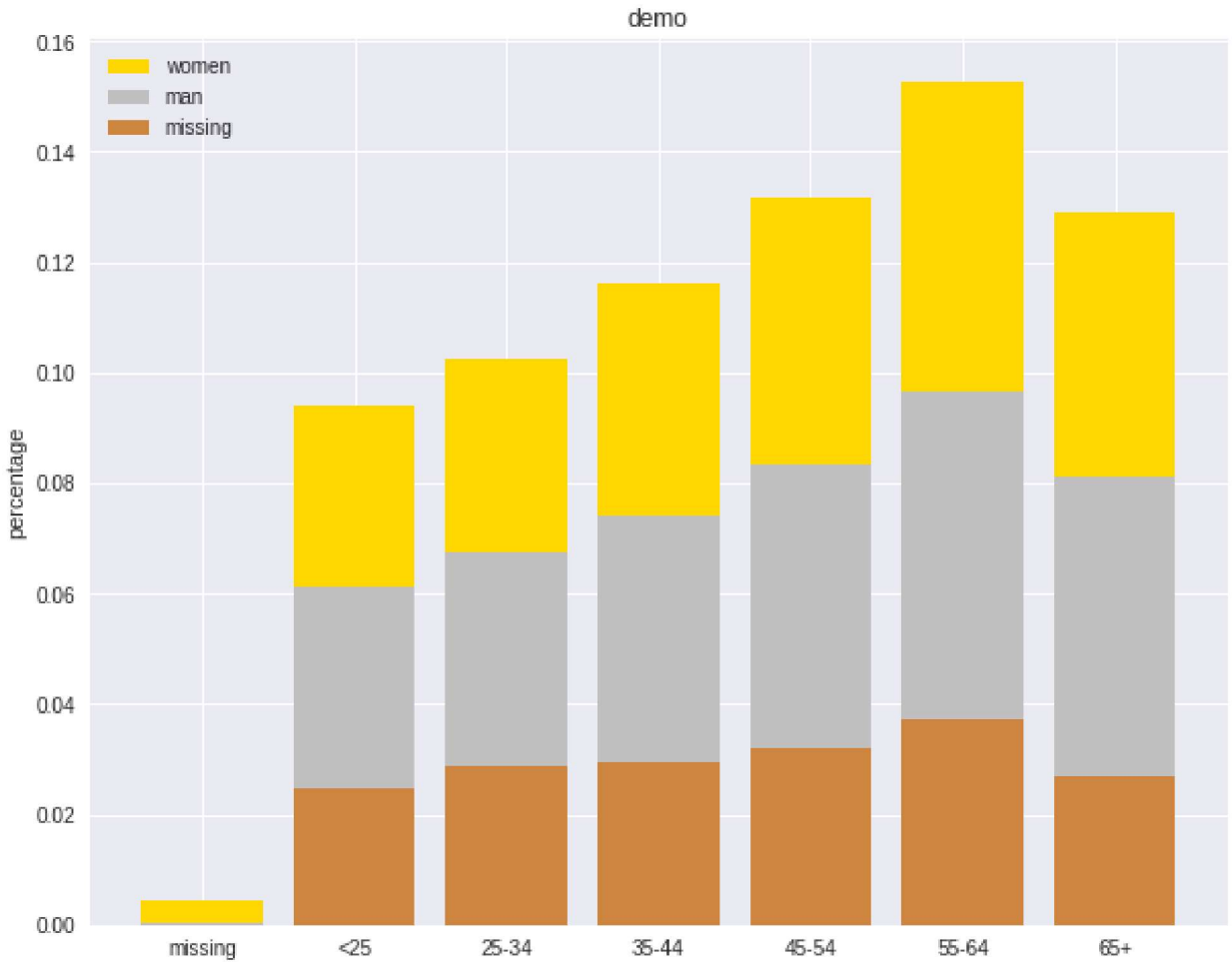


Bar plot

```
labels = plot_data.age_class
missing = plot_data.Percent
ind = [x for x, _ in enumerate(labels)]

plt.figure(figsize=(10,8))
plt.bar(ind, missing, width=0.8, label='missing', color='gold')

plt.xticks(ind, labels)
plt.ylabel("percentage")

plt.show()
```




```
labels = ['missing', '<25', '25-34', '35-44', '45-54','55-64','65+']
missing = np.array([0.000095, 0.024830, 0.028665, 0.029477, 0.031918,0.037073,
↪0.026699])
man = np.array([0.000147, 0.036311, 0.038684, 0.044761, 0.051269, 0.059542, 0.
↪054259])
women = np.array([0.004035, 0.032935, 0.035351, 0.041778, 0.048437, 0.056236,
↪0.048091])
ind = [x for x, _ in enumerate(labels)]

plt.figure(figsize=(10,8))
plt.bar(ind, women, width=0.8, label='women', color='gold',␣
↪bottom=man+missing)
plt.bar(ind, man, width=0.8, label='man', color='silver', bottom=missing)
plt.bar(ind, missing, width=0.8, label='missing', color='#CD853F')

plt.xticks(ind, labels)
plt.ylabel("percentage")
plt.legend(loc="upper left")
plt.title("demo")

plt.show()
```



7.2 Multivariate Analysis

In this section, I will only demostrate the bivariate analysis. Since the multivariate analysis is the generation of the bivariate.

7.2.1 Numerical V.S. Numerical

Correlation matrix

```
from pyspark.mllib.stat import Statistics
import pandas as pd

corr_data = df.select(num_cols)

col_names = corr_data.columns
features = corr_data.rdd.map(lambda row: row[0:])
corr_mat=Statistics.corr(features, method="pearson")
```

(continues on next page)

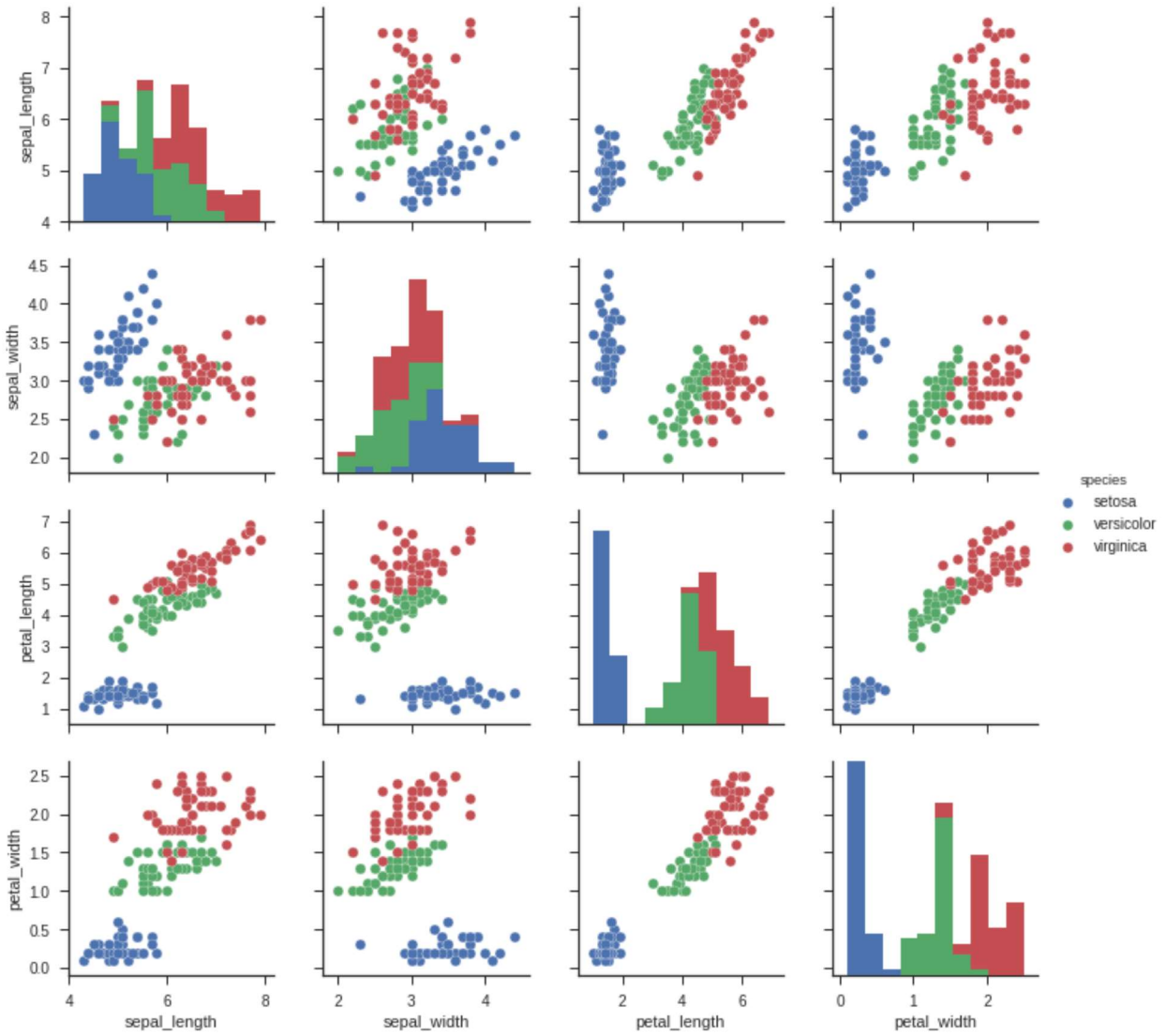
(continued from previous page)

```
corr_df = pd.DataFrame(corr_mat)
corr_df.index, corr_df.columns = col_names, col_names

print(corr_df.to_string())
```

```
+--------------------+--------------------+
|     Account Balance|    No of dependents|
+--------------------+--------------------+
|                 1.0|-0.01414542650320914|
|-0.01414542650320914|                 1.0|
+--------------------+--------------------+
```

Scatter Plot

```
import seaborn as sns
sns.set(style="ticks")

df = sns.load_dataset("iris")
sns.pairplot(df, hue="species")
plt.show()
```



7.2.2 Categorical V.S. Categorical

Pearson's Chi-squared test

Warning: `pyspark.ml.stat` is only available in Spark 2.4.0.

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.stat import ChiSquareTest

data = [(0.0, Vectors.dense(0.5, 10.0)),
        (0.0, Vectors.dense(1.5, 20.0)),
        (1.0, Vectors.dense(1.5, 30.0)),
        (0.0, Vectors.dense(3.5, 30.0)),
```

(continues on next page)

(continued from previous page)

```
        (0.0, Vectors.dense(3.5, 40.0)),
        (1.0, Vectors.dense(3.5, 40.0))]
df = spark.createDataFrame(data, ["label", "features"])

r = ChiSquareTest.test(df, "features", "label").head()
print("pValues: " + str(r.pValues))
print("degreesOfFreedom: " + str(r.degreesOfFreedom))
print("statistics: " + str(r.statistics))
```

```
pValues: [0.687289278791,0.682270330336]
degreesOfFreedom: [2, 3]
statistics: [0.75,1.5]
```

Cross table

```
df.stat.crosstab("age_class", "Occupation").show()
```

```
+--------------------+---+---+---+---+
|age_class_Occupation|  1|  2|  3|  4|
+--------------------+---+---+---+---+
|                 <25|  4| 34|108|  4|
|               55-64|  1| 15| 31|  9|
|               25-34|  7| 61|269| 60|
|               35-44|  4| 58|143| 49|
|                 65+|  5|  3|  6|  9|
|               45-54|  1| 29| 73| 17|
+--------------------+---+---+---+---+
```

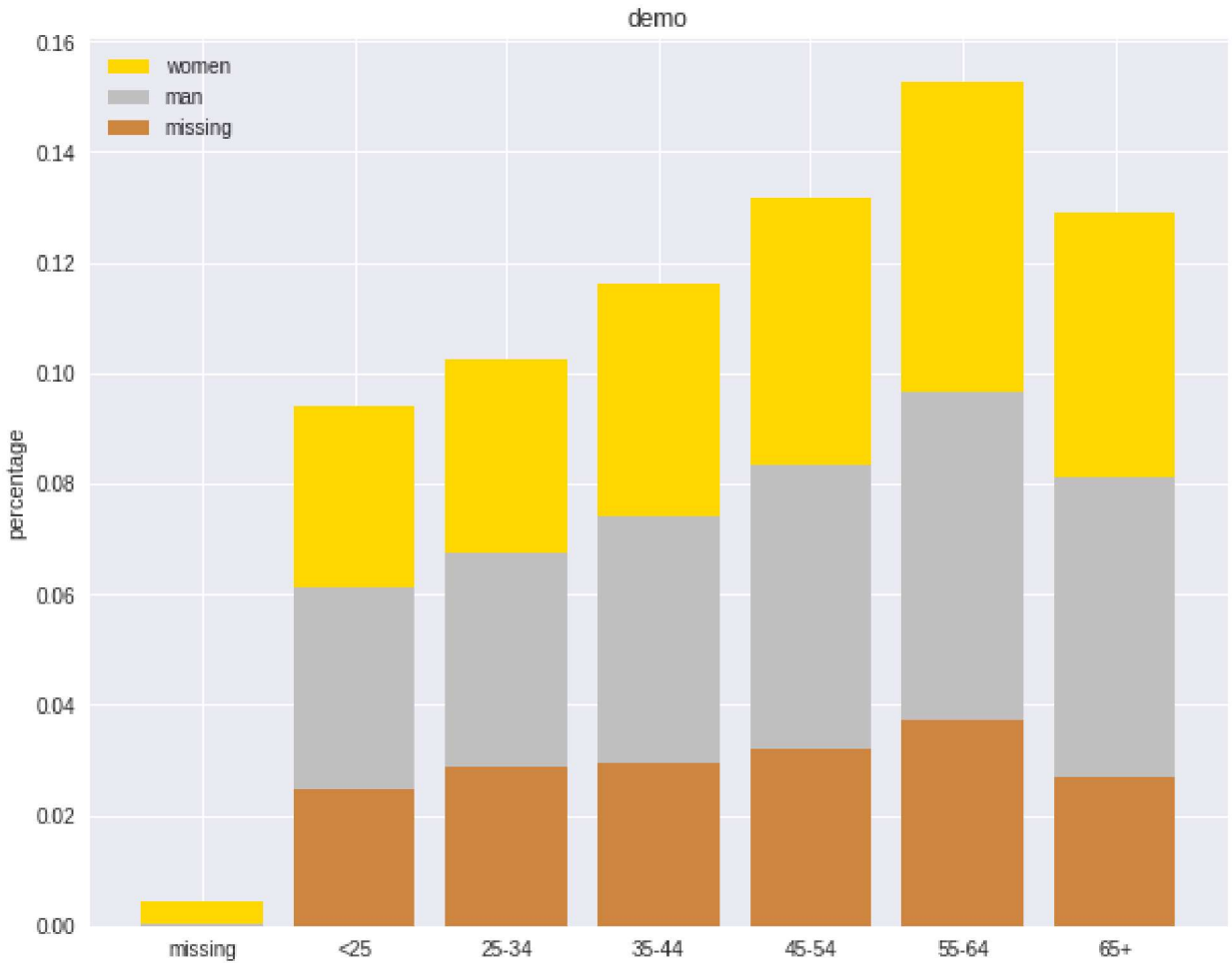
Stacked plot

```
labels = ['missing', '<25', '25-34', '35-44', '45-54','55-64','65+']
missing = np.array([0.000095, 0.024830, 0.028665, 0.029477, 0.031918,0.037073,
↪0.026699])
man = np.array([0.000147, 0.036311, 0.038684, 0.044761, 0.051269, 0.059542, 0.
↪054259])
women = np.array([0.004035, 0.032935, 0.035351, 0.041778, 0.048437, 0.056236,
↪0.048091])
ind = [x for x, _ in enumerate(labels)]

plt.figure(figsize=(10,8))
plt.bar(ind, women, width=0.8, label='women', color='gold',␣
↪bottom=man+missing)
plt.bar(ind, man, width=0.8, label='man', color='silver', bottom=missing)
plt.bar(ind, missing, width=0.8, label='missing', color='#CD853F')

plt.xticks(ind, labels)
plt.ylabel("percentage")
```

(continues on next page)

(continued from previous page)

```
plt.legend(loc="upper left")
plt.title("demo")

plt.show()
```



7.2.3 Numerical V.S. Categorical

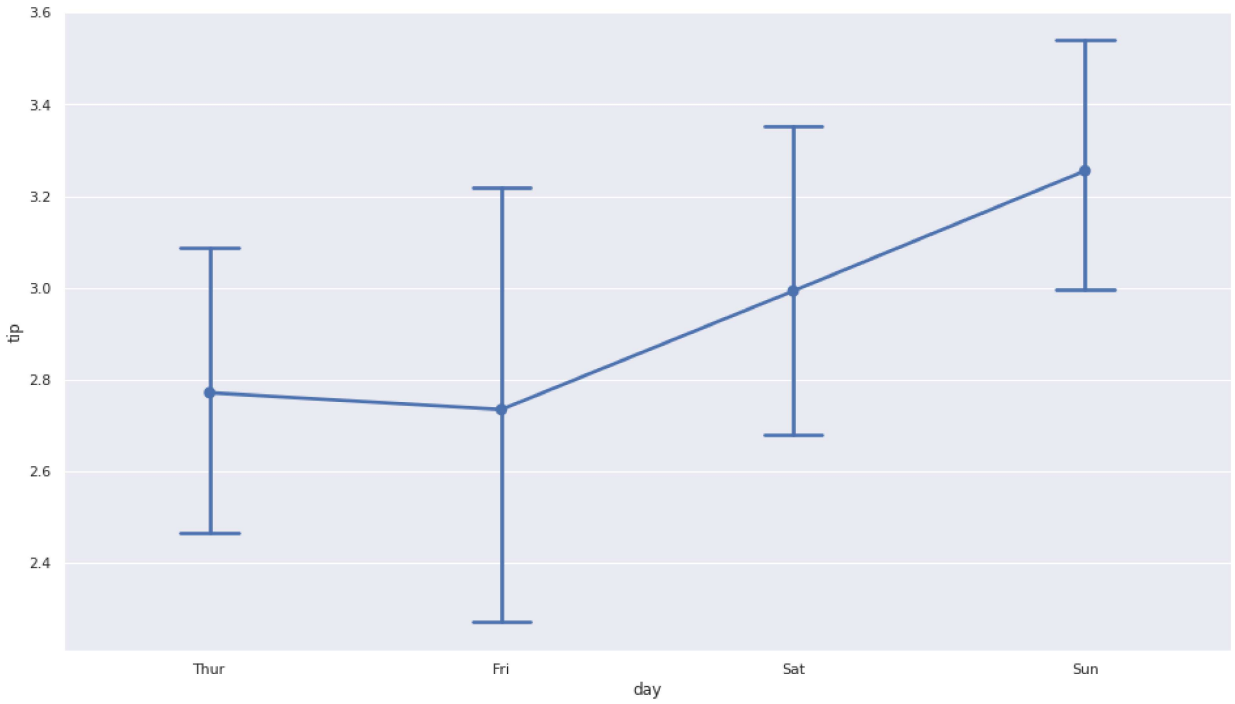
Line Chart with Error Bars

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
%matplotlib inline

plt.rcParams['figure.figsize'] =(16,9)
plt.style.use('ggplot')
```

(continues on next page)

(continued from previous page)

```
sns.set()

ax = sns.pointplot(x="day", y="tip", data=tips, capsize=.2)
plt.show()
```



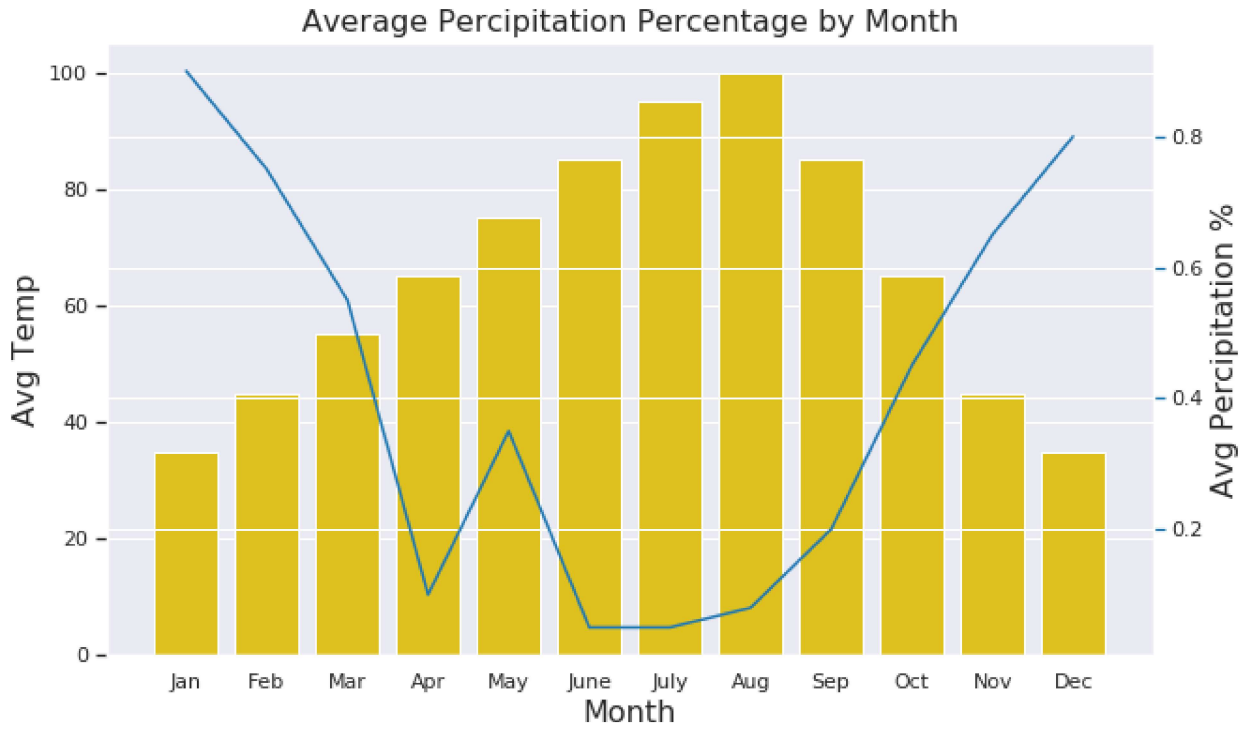
Combination Chart

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
%matplotlib inline

plt.rcParams['figure.figsize'] =(16,9)
plt.style.use('ggplot')
sns.set()


#create list of months
Month = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June',
         'July', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
#create list for made up average temperatures
Avg_Temp = [35, 45, 55, 65, 75, 85, 95, 100, 85, 65, 45, 35]
#create list for made up average percipitation %
Avg_Percipitation_Perc = [.90, .75, .55, .10, .35, .05, .05, .08, .20, .45, .
↪65, .80]
```

(continues on next page)

(continued from previous page)

```
#assign lists to a value
data = {'Month': Month, 'Avg_Temp': Avg_Temp, 'Avg_Percipitation_Perc': Avg_
↪Percipitation_Perc}
#convert dictionary to a dataframe
df = pd.DataFrame(data)


fig, ax1 = plt.subplots(figsize=(10,6))
ax1.set_title('Average Percipitation Percentage by Month', fontsize=16)
ax1.tick_params(axis='y')

ax2 = sns.barplot(x='Month', y='Avg_Temp', data = df, color = 'gold')
ax2 = ax1.twinx()
ax2 = sns.lineplot(x='Month', y='Avg_Percipitation_Perc', data = df,␣
↪sort=False, color=color)

ax1.set_xlabel('Month', fontsize=16)
ax1.set_ylabel('Avg Temp', fontsize=16)

ax2.tick_params(axis='y', color=color)
ax2.set_ylabel('Avg Percipitation %', fontsize=16)
plt.show()
```



CHAPTER

EIGHT

DATA MANIPULATION: FEATURES

Chinese proverb

All things are diffcult before they are easy!

Feature building is a super important step for modeling which will determine the success or failure of your model. Otherwise, you will get: garbage in; garbage out! The techniques have been covered in the following chapters, the followings are the brief summary. I recently found that the Spark official website did a really good job for tutorial documentation. The chapter is based on Extracting transforming and selecting features.

8.1 Feature Extraction

8.1.1 TF-IDF

Term frequency-inverse document frequency (TF-IDF) is a feature vectorization method widely used in text mining to reflect the importance of a term to a document in the corpus. More details can be found at: https://spark.apache.org/docs/latest/ml-features#feature-extractors

Stackoverflow TF: Both HashingTF and CountVectorizer can be used to generate the term frequency vectors. A few important differences:

a. partially **reversible** (CountVectorizer) vs **irreversible** (HashingTF) - since hashing is not reversible you cannot restore original input from a hash vector. From the other hand count vector with model (index) can be used to restore unordered input. As a consequence models created using hashed input can be much harder to interpret and monitor.

b. **memory and computational overhead** - HashingTF requires only a single data scan and no additional memory beyond original input and vector. CountVectorizer requires additional scan over the data to build a model and additional memory to store vocabulary (index). In case of unigram language model it is usually not a problem but in case of higher n-grams it can be prohibitively expensive or not feasible.

c. hashing **depends on** a size of the vector , hashing function and a document. Counting depends on a size of the vector, training corpus and a document.

d. **a source of the information loss** - in case of HashingTF it is dimensionality reduction with possible collisions. CountVectorizer discards infrequent tokens. How it affects downstream models depends on a particular use case and data.

HashingTF and **CountVectorizer** are the two popular alogoritms which used to generate term frequency vectors. They basically convert documents into a numerical representation which can be fed directly or with further processing into other algorithms like LDA, MinHash for Jaccard Distance, Cosine Distance.

- t: term
- d: document
- D: corpus
- $|D|$: the number of the elements in corpus
- $TF(t, d)$: Term Frequency: the number of times that term t appears in document d
- $DF(t, D)$: Document Frequency: the number of documents that contains term t
- $IDF(t, D)$: Inverse Document Frequency is a numerical measure of how much information a term provides

$$IDF(t, D) = \log \frac{|D| + 1}{DF(t, D) + 1}$$

- $TFIDF(t, d, D)$ the product of TF and IDF

$$TFIDF(t, d, D) = TF(t, d) \cdot IDF(t, D)$$

Let's look at the example:

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import HashingTF, IDF, Tokenizer

sentenceData = spark.createDataFrame([
    (0, "Python python Spark Spark"),
    (1, "Python SQL")],
 ["document", "sentence"])
```

```
sentenceData.show(truncate=False)
+--------+-------------------------+
|document|sentence                 |
+--------+-------------------------+
|0       |Python python Spark Spark|
|1       |Python SQL               |
+--------+-------------------------+
```

Then:

- $TF(python, document1) = 1, TF(spark, document1) = 2$
- $DF(Spark, D) = 2, DF(sql, D) = 1$
- IDF:

$$IDF(python, D) = \log \frac{|D|+1}{DF(t,D)+1} = \log(\frac{2+1}{2+1}) = 0$$

$$IDF(spark, D) = \log \frac{|D|+1}{DF(t,D)+1} = \log(\frac{2+1}{1+1}) = 0.4054651081081644$$

$$IDF(sql, D) = \log \frac{|D|+1}{DF(t,D)+1} = \log(\frac{2+1}{1+1}) = 0.4054651081081644$$

- TFIDF

$$TFIDF(python, document1, D) = 3 * 0 = 0$$

$$TFIDF(spark, document1, D) = 2 * 0.4054651081081644 = 0.8109302162163288$$

$$TFIDF(sql, document1, D) = 1 * 0.4054651081081644 = 0.4054651081081644$$

Countvectorizer

Stackoverflow TF: CountVectorizer and CountVectorizerModel aim to help convert a collection of text documents to vectors of token counts. When an a-priori dictionary is not available, CountVectorizer can be used as an Estimator to extract the vocabulary, and generates a CountVectorizerModel. The model produces sparse representations for the documents over the vocabulary, which can then be passed to other algorithms like LDA.

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import CountVectorizer
from pyspark.ml.feature import HashingTF, IDF, Tokenizer

sentenceData = spark.createDataFrame([
    (0, "Python python Spark Spark"),
    (1, "Python SQL")],
 ["document", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
vectorizer  = CountVectorizer(inputCol="words", outputCol="rawFeatures")

idf = IDF(inputCol="rawFeatures", outputCol="features")

pipeline = Pipeline(stages=[tokenizer, vectorizer, idf])

model = pipeline.fit(sentenceData)
```

```
import numpy as np

total_counts = model.transform(sentenceData)\
                    .select('rawFeatures').rdd\
                    .map(lambda row: row['rawFeatures'].toArray())\
                    .reduce(lambda x,y: [x[i]+y[i] for i in range(len(y))])

vocabList = model.stages[1].vocabulary
d = {'vocabList':vocabList,'counts':total_counts}
```

(continues on next page)

(continued from previous page)

```
spark.createDataFrame(np.array(list(d.values())).T.tolist(),list(d.keys())).
↪show()
```

```
counts = model.transform(sentenceData).select('rawFeatures').collect()
counts

[Row(rawFeatures=SparseVector(8, {0: 1.0, 1: 1.0, 2: 1.0})),
 Row(rawFeatures=SparseVector(8, {0: 1.0, 1: 1.0, 4: 1.0})),
 Row(rawFeatures=SparseVector(8, {0: 1.0, 3: 1.0, 5: 1.0, 6: 1.0, 7: 1.0}))]
```

```
+---------+------+
|vocabList|counts|
+---------+------+
|   python|   3.0|
|    spark|   2.0|
|      sql|   1.0|
+---------+------+
```

```
model.transform(sentenceData).show(truncate=False)
```

```
+--------+-------------------------+------------------------------+-----------
↪--------+----------------------------------+
|document|sentence                 |words
↪|rawFeatures        |features                          |
+--------+-------------------------+------------------------------+-----------
↪--------+----------------------------------+
|0       |Python python Spark Spark|[python, python, spark, spark]|(3,[0,1],
↪[2.0,2.0])|(3,[0,1],[0.0,0.8109302162163288])|
|1       |Python SQL               |[python, sql]                 |(3,[0,2],
↪[1.0,1.0])|(3,[0,2],[0.0,0.4054651081081644])|
+--------+-------------------------+------------------------------+-----------
↪--------+----------------------------------+
```

```
from pyspark.sql.types import ArrayType, StringType

def termsIdx2Term(vocabulary):
    def termsIdx2Term(termIndices):
        return [vocabulary[int(index)] for index in termIndices]
    return udf(termsIdx2Term, ArrayType(StringType()))

vectorizerModel = model.stages[1]
vocabList = vectorizerModel.vocabulary
vocabList
```

```
['python', 'spark', 'sql']
```

```
rawFeatures = model.transform(sentenceData).select('rawFeatures')
rawFeatures.show()
```

(continues on next page)

(continued from previous page)

```
+--------------------+
|         rawFeatures|
+--------------------+
|(3,[0,1],[2.0,2.0])|
|(3,[0,2],[1.0,1.0])|
+--------------------+
```

```
from pyspark.sql.functions import udf
import pyspark.sql.functions as F
from pyspark.sql.types import  StringType, DoubleType, IntegerType

indices_udf = udf(lambda vector: vector.indices.tolist(),␣
↪ArrayType(IntegerType()))
values_udf = udf(lambda vector: vector.toArray().tolist(),␣
↪ArrayType(DoubleType()))


rawFeatures.withColumn('indices', indices_udf(F.col('rawFeatures')))\
           .withColumn('values', values_udf(F.col('rawFeatures')))\
           .withColumn("Terms", termsIdx2Term(vocabList)("indices")).show()
```

```
+--------------------+-------+---------------+---------------+
|         rawFeatures|indices|         values|          Terms|
+--------------------+-------+---------------+---------------+
|(3,[0,1],[2.0,2.0])| [0, 1]|[2.0, 2.0, 0.0]|[python, spark]|
|(3,[0,2],[1.0,1.0])| [0, 2]|[1.0, 0.0, 1.0]|  [python, sql]|
+--------------------+-------+---------------+---------------+
```

HashingTF

Stackoverflow TF: HashingTF is a Transformer which takes sets of terms and converts those sets into fixed-length feature vectors. In text processing, a "set of terms" might be a bag of words. HashingTF utilizes the hashing trick. A raw feature is mapped into an index (term) by applying a hash function. The hash function used here is MurmurHash 3. Then term frequencies are calculated based on the mapped indices. This approach avoids the need to compute a global term-to-index map, which can be expensive for a large corpus, but it suffers from potential hash collisions, where different raw features may become the same term after hashing.

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import HashingTF, IDF, Tokenizer

sentenceData = spark.createDataFrame([
    (0, "Python python Spark Spark"),
    (1, "Python SQL")],
 ["document", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
```

(continues on next page)

(continued from previous page)

```
vectorizer  = HashingTF(inputCol="words", outputCol="rawFeatures",␣
↪numFeatures=5)

idf = IDF(inputCol="rawFeatures", outputCol="features")

pipeline = Pipeline(stages=[tokenizer, vectorizer, idf])


model = pipeline.fit(sentenceData)
model.transform(sentenceData).show(truncate=False)
```

```
+--------+-------------------------+------------------------------+-----------
↪--------+---------------------------------+
|document|sentence                 |words                         ␣
↪|rawFeatures        |features                         |
+--------+-------------------------+------------------------------+-----------
↪--------+---------------------------------+
|0       |Python python Spark Spark|[python, python, spark, spark]|(5,[0,4],
↪[2.0,2.0])|(5,[0,4],[0.8109302162163288,0.0])|
|1       |Python SQL               |[python, sql]                 |(5,[1,4],
↪[1.0,1.0])|(5,[1,4],[0.4054651081081644,0.0])|
+--------+-------------------------+------------------------------+-----------
↪--------+---------------------------------+
```

8.1.2 Word2Vec

Word Embeddings

Word2Vec is one of the popupar method to implement the **Word Embeddings**. Word embeddings (The best tutorial I have read. The following word and images content are from Chris Bail, PhD Duke University. So the copyright belongs to Chris Bail, PhD Duke University.) gained fame in the world of automated text analysis when it was demonstrated that they could be used to identify analogies. Figure 1 illustrates the output of a word embedding model where individual words are plotted in three dimensional space generated by the model. By examining the adjacency of words in this space, word embedding models can complete analogies such as "Man is to woman as king is to queen." If you'd like to explore what the output of a large word embedding model looks like in more detail, check out this fantastic visualization of most words in the English language that was produced using a word embedding model called GloVE.

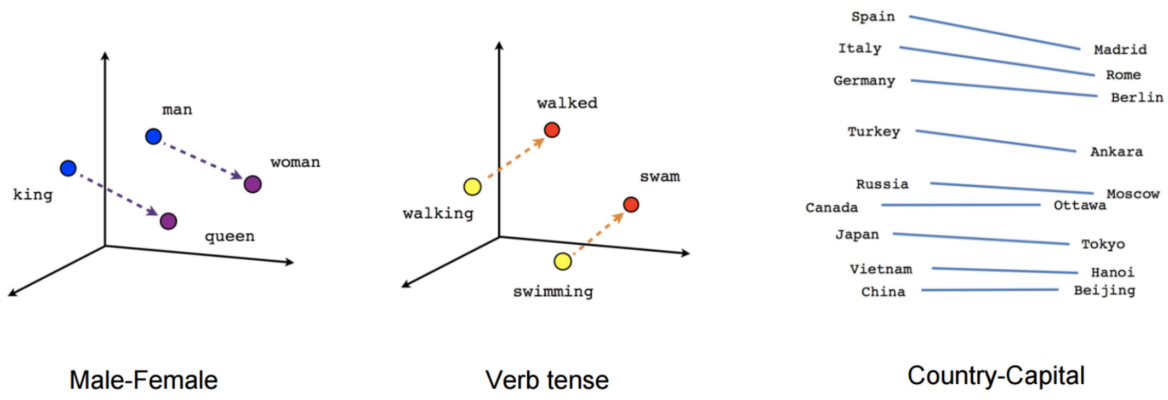


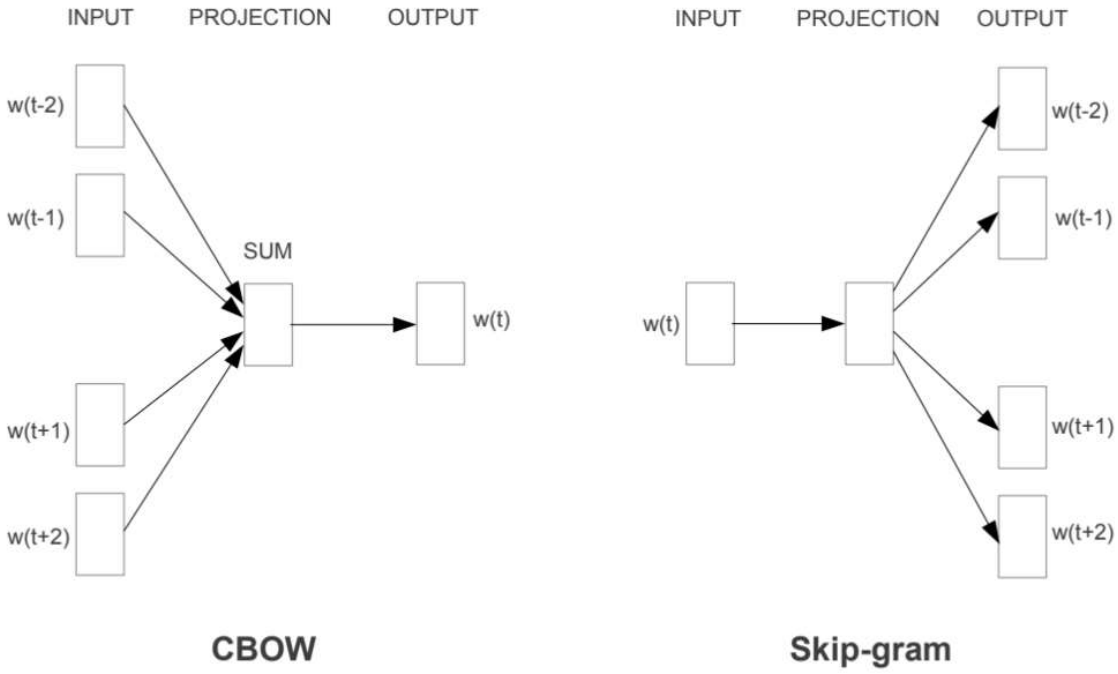
Fig. 1: output of a word embedding model

The Context Window

Word embeddings are created by identifying the words that occur within something called a "Context Window." The Figure below illustrates context windows of varied length for a single sentence. The context window is defined by a string of words before and after a focal or "center" word that will be used to train a word embedding model. Each center word and context words can be represented as a vector of numbers that describe the presence or absence of unique words within a dataset, which is perhaps why word embedding models are often described as "word vector" models, or "word2vec" models.

Two Types of Embedding Models

Word embeddings are usually performed in one of two ways: "Continuous Bag of Words" (CBOW) or a "Skip-Gram Model." The figure below illustrates the differences between the two models. The CBOW model reads in the context window words and tries to predict the most likely center word. The Skip-Gram Model predicts the context words given the center word. The examples above were created using the Skip-Gram model, which is perhaps most useful for people who want to identify patterns within texts to represent them in multimensional space, whereas the CBOW model is more useful in practical applications such as predictive web search.



Word Embedding Models in PySpark

```
from pyspark.ml.feature import Word2Vec

from pyspark.ml import Pipeline

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol="words", outputCol=
↪"feature")

pipeline = Pipeline(stages=[tokenizer, word2Vec])


model = pipeline.fit(sentenceData)
result = model.transform(sentenceData)
```

```
result.show()
+-----+--------------------+--------------------+--------------------+
|label|            sentence|               words|             feature|
+-----+--------------------+--------------------+--------------------+
|  0.0|        I love Spark|    [i, love, spark]|[0.05594437588782...|
|  0.0|       I love python|   [i, love, python]|[-0.0350368790871...|
|  1.0|I think ML is awe...|[i, think, ml, is...|[0.01242086507845...|
+-----+--------------------+--------------------+--------------------+
```

```
w2v = model.stages[1]
w2v.getVectors().show()
```

(continues on next page)

(continued from previous page)

```
+-------+-----------------------------------------------------------------+
|word   |vector                                                           |
+-------+-----------------------------------------------------------------+
|is     |[0.13657838106155396,0.060924094170331955,-0.03379475697875023]  |
|awesome|[0.037024181336164474,-0.023855900391936302,0.07600378245115528] |
|i      |[-0.0014482572441920638,0.049365971237421036,0.12016955763101578]|
|ml     |[-0.14006119966506958,0.01626444421708584,0.042281970381736755]  |
|spark  |[0.1589149385690689,-0.10970081388950348,-0.10547549277544022]   |
|think  |[0.030011219903826714,-0.08994936943054199,0.16471518576145172]  |
|love   |[0.01036644633859396,-0.017782460898160934,0.08870164304971695]  |
|python |[-0.11402882635593414,0.045119188725948334,-0.02987742377467155]|
+-------+-----------------------------------------------------------------+
```

```
from pyspark.sql.functions import format_number as fmt
w2v.findSynonyms("could", 2).select("word", fmt("similarity", 5).alias(
↪"similarity")).show()
```

```
+-------+----------+
|   word|similarity|
+-------+----------+
|classes|   0.90232|
|      i|   0.75424|
+-------+----------+
```

8.1.3 FeatureHasher

```
from pyspark.ml.feature import FeatureHasher

dataset = spark.createDataFrame([
    (2.2, True, "1", "foo"),
    (3.3, False, "2", "bar"),
    (4.4, False, "3", "baz"),
    (5.5, False, "4", "foo")
], ["real", "bool", "stringNum", "string"])

hasher = FeatureHasher(inputCols=["real", "bool", "stringNum", "string"],
                       outputCol="features")

featurized = hasher.transform(dataset)
featurized.show(truncate=False)
```

```
+----+-----+---------+------+-----------------------------------------------------
↪-------+
|real|bool |stringNum|string|features
↪       |
+----+-----+---------+------+-----------------------------------------------------
↪-------+
|2.2 |true |1        |foo   |(262144,[174475,247670,257907,262126],[2.2,1.0,1.
↪0,1.0])|
```

(continues on next page)

(continued from previous page)

```
|3.3 |false|2         |bar    |(262144,[70644,89673,173866,174475],[1.0,1.0,1.0,
↪3.3])   |
|4.4 |false|3         |baz    |(262144,[22406,70644,174475,187923],[1.0,1.0,4.4,
↪1.0])   |
|5.5 |false|4         |foo    |(262144,[70644,101499,174475,257907],[1.0,1.0,5.
↪5,1.0]) |
+----+-----+----------+-------+-------------------------------------------------
↪-------+
```

8.1.4 RFormula

```
from pyspark.ml.feature import RFormula

dataset = spark.createDataFrame(
    [(7, "US", 18, 1.0),
     (8, "CA", 12, 0.0),
     (9, "CA", 15, 0.0)],
    ["id", "country", "hour", "clicked"])

formula = RFormula(
    formula="clicked ~ country + hour",
    featuresCol="features",
    labelCol="label")

output = formula.fit(dataset).transform(dataset)
output.select("features", "label").show()
```

```
+----------+-----+
|  features|label|
+----------+-----+
|[0.0,18.0]|  1.0|
|[1.0,12.0]|  0.0|
|[1.0,15.0]|  0.0|
+----------+-----+
```

8.2 Feature Transform

8.2.1 Tokenizer

```
from pyspark.ml.feature import Tokenizer, RegexTokenizer
from pyspark.sql.functions import col, udf
from pyspark.sql.types import IntegerType

sentenceDataFrame = spark.createDataFrame([
    (0, "Hi I heard about Spark"),
    (1, "I wish Java could use case classes"),
```

(continues on next page)

(continued from previous page)

```
    (2, "Logistic,regression,models,are,neat")
], ["id", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")

regexTokenizer = RegexTokenizer(inputCol="sentence", outputCol="words",␣
↪pattern="\\W")
# alternatively, pattern="\\w+", gaps(False)

countTokens = udf(lambda words: len(words), IntegerType())

tokenized = tokenizer.transform(sentenceDataFrame)
tokenized.select("sentence", "words")\
    .withColumn("tokens", countTokens(col("words"))).show(truncate=False)

regexTokenized = regexTokenizer.transform(sentenceDataFrame)
regexTokenized.select("sentence", "words") \
    .withColumn("tokens", countTokens(col("words"))).show(truncate=False)
```

```
+-----------------------------------+-----------------------------------------
↪-+------+
|sentence                           |words                                    ␣
↪ |tokens|
+-----------------------------------+-----------------------------------------
↪-+------+
|Hi I heard about Spark             |[hi, i, heard, about, spark]             ␣
↪ |5     |
|I wish Java could use case classes |[i, wish, java, could, use, case,␣
↪classes]|7     |
|Logistic,regression,models,are,neat|[logistic,regression,models,are,neat]    ␣
↪ |1     |
+-----------------------------------+-----------------------------------------
↪-+------+

+-----------------------------------+-----------------------------------------
↪-+------+
|sentence                           |words                                    ␣
↪ |tokens|
+-----------------------------------+-----------------------------------------
↪-+------+
|Hi I heard about Spark             |[hi, i, heard, about, spark]             ␣
↪ |5     |
|I wish Java could use case classes |[i, wish, java, could, use, case,␣
↪classes]|7     |
|Logistic,regression,models,are,neat|[logistic, regression, models, are,␣
↪neat] |5     |
+-----------------------------------+-----------------------------------------
↪-+------+
```

8.2.2 StopWordsRemover

```
from pyspark.ml.feature import StopWordsRemover

sentenceData = spark.createDataFrame([
    (0, ["I", "saw", "the", "red", "balloon"]),
    (1, ["Mary", "had", "a", "little", "lamb"])
], ["id", "raw"])

remover = StopWordsRemover(inputCol="raw", outputCol="removeded")
remover.transform(sentenceData).show(truncate=False)
```

```
+---+----------------------------+--------------------+
|id |raw                         |removeded           |
+---+----------------------------+--------------------+
|0  |[I, saw, the, red, balloon] |[saw, red, balloon] |
|1  |[Mary, had, a, little, lamb]|[Mary, little, lamb]|
+---+----------------------------+--------------------+
```

8.2.3 NGram

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import CountVectorizer
from pyspark.ml.feature import HashingTF, IDF, Tokenizer

from pyspark.ml.feature import NGram

sentenceData = spark.createDataFrame([
    (0.0, "I love Spark"),
    (0.0, "I love python"),
    (1.0, "I think ML is awesome")],
 ["label", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
ngram = NGram(n=2, inputCol="words", outputCol="ngrams")

idf = IDF(inputCol="rawFeatures", outputCol="features")

pipeline = Pipeline(stages=[tokenizer, ngram])

model = pipeline.fit(sentenceData)

model.transform(sentenceData).show(truncate=False)
```

```
+-----+--------------------+--------------------------+----------------------
↪-----------------+
|label|sentence            |words                     |ngrams                ␣
↪                  |
+-----+--------------------+--------------------------+----------------------
↪-----------------+
```

(continues on next page)

(continued from previous page)

```
|0.0  |I love Spark          |[i, love, spark]           |[i love, love spark]␣
↪                  |
|0.0  |I love python         |[i, love, python]          |[i love, love␣
↪python]                  |
|1.0  |I think ML is awesome|[i, think, ml, is, awesome]|[i think, think ml,␣
↪ml is, is awesome]|
+-----+---------------------+---------------------------+--------------------
↪-----------------+
```

8.2.4 Binarizer

```
from pyspark.ml.feature import Binarizer

continuousDataFrame = spark.createDataFrame([
    (0, 0.1),
    (1, 0.8),
    (2, 0.2),
    (3,0.5)
], ["id", "feature"])

binarizer = Binarizer(threshold=0.5, inputCol="feature", outputCol="binarized_
↪feature")

binarizedDataFrame = binarizer.transform(continuousDataFrame)

print("Binarizer output with Threshold = %f" % binarizer.getThreshold())
binarizedDataFrame.show()
```

```
Binarizer output with Threshold = 0.500000
+---+-------+-----------------+
| id|feature|binarized_feature|
+---+-------+-----------------+
|  0|    0.1|              0.0|
|  1|    0.8|              1.0|
|  2|    0.2|              0.0|
|  3|    0.5|              0.0|
+---+-------+-----------------+
```

8.2.5 Bucketizer

[Bucketizer](https://spark.apache.org/docs/latest/ml-features.html#bucketizer) transforms a column of continuous features to a column of feature buckets, where the buckets are specified by users.

```
from pyspark.ml.feature import QuantileDiscretizer, Bucketizer

data = [(0, 18.0), (1, 19.0), (2, 8.0), (3, 5.0), (4, 2.0)]
df = spark.createDataFrame(data, ["id", "age"])
```

(continues on next page)

(continued from previous page)

```
print(df.show())

splits = [-float("inf"),3, 10,float("inf")]
result_bucketizer = Bucketizer(splits=splits, inputCol="age",outputCol="result
↪").transform(df)
result_bucketizer.show()
```

```
+---+----+
| id| age|
+---+----+
|  0|18.0|
|  1|19.0|
|  2| 8.0|
|  3| 5.0|
|  4| 2.0|
+---+----+

None
+---+----+------+
| id| age|result|
+---+----+------+
|  0|18.0|   2.0|
|  1|19.0|   2.0|
|  2| 8.0|   1.0|
|  3| 5.0|   1.0|
|  4| 2.0|   0.0|
+---+----+------+
```

8.2.6 QuantileDiscretizer

QuantileDiscretizer takes a column with continuous features and outputs a column with binned categorical features. The number of bins is set by the numBuckets parameter. It is possible that the number of buckets used will be smaller than this value, for example, if there are too few distinct values of the input to create enough distinct quantiles.

```
from pyspark.ml.feature import QuantileDiscretizer, Bucketizer

data = [(0, 18.0), (1, 19.0), (2, 8.0), (3, 5.0), (4, 2.0)]
df = spark.createDataFrame(data, ["id", "age"])
print(df.show())

qds = QuantileDiscretizer(numBuckets=5, inputCol="age", outputCol="buckets",
                          relativeError=0.01, handleInvalid="error")
bucketizer = qds.fit(df)
bucketizer.transform(df).show()
bucketizer.setHandleInvalid("skip").transform(df).show()
```

```
+---+----+
| id| age|
```

(continues on next page)

(continued from previous page)

```
+---+----+
|  0|18.0|
|  1|19.0|
|  2| 8.0|
|  3| 5.0|
|  4| 2.0|
+---+----+

None
+---+----+-------+
| id| age|buckets|
+---+----+-------+
|  0|18.0|    3.0|
|  1|19.0|    3.0|
|  2| 8.0|    2.0|
|  3| 5.0|    2.0|
|  4| 2.0|    1.0|
+---+----+-------+

+---+----+-------+
| id| age|buckets|
+---+----+-------+
|  0|18.0|    3.0|
|  1|19.0|    3.0|
|  2| 8.0|    2.0|
|  3| 5.0|    2.0|
|  4| 2.0|    1.0|
+---+----+-------+
```

If the data has NULL values, then you will get the following results:

```
from pyspark.ml.feature import QuantileDiscretizer, Bucketizer

data = [(0, 18.0), (1, 19.0), (2, 8.0), (3, 5.0), (4, None)]
df = spark.createDataFrame(data, ["id", "age"])
print(df.show())

splits = [-float("inf"),3, 10,float("inf")]
result_bucketizer = Bucketizer(splits=splits,
                               inputCol="age",outputCol="result").
↪transform(df)
result_bucketizer.show()

qds = QuantileDiscretizer(numBuckets=5, inputCol="age", outputCol="buckets",
                               relativeError=0.01, handleInvalid="error")
bucketizer = qds.fit(df)
bucketizer.transform(df).show()
bucketizer.setHandleInvalid("skip").transform(df).show()
```

```
+---+----+
| id| age|
```

(continues on next page)

(continued from previous page)

```
+---+----+
|  0|18.0|
|  1|19.0|
|  2| 8.0|
|  3| 5.0|
|  4|null|
+---+----+

None
+---+----+------+
| id| age|result|
+---+----+------+
|  0|18.0|   2.0|
|  1|19.0|   2.0|
|  2| 8.0|   1.0|
|  3| 5.0|   1.0|
|  4|null|  null|
+---+----+------+

+---+----+-------+
| id| age|buckets|
+---+----+-------+
|  0|18.0|    3.0|
|  1|19.0|    4.0|
|  2| 8.0|    2.0|
|  3| 5.0|    1.0|
|  4|null|   null|
+---+----+-------+

+---+----+-------+
| id| age|buckets|
+---+----+-------+
|  0|18.0|    3.0|
|  1|19.0|    4.0|
|  2| 8.0|    2.0|
|  3| 5.0|    1.0|
+---+----+-------+
```

8.2.7 StringIndexer

```
from pyspark.ml.feature import StringIndexer

df = spark.createDataFrame(
    [(0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c")],
    ["id", "category"])

indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")
indexed = indexer.fit(df).transform(df)
indexed.show()
```

```
+---+--------+-------------+
| id|category|categoryIndex|
+---+--------+-------------+
|  0|       a|          0.0|
|  1|       b|          2.0|
|  2|       c|          1.0|
|  3|       a|          0.0|
|  4|       a|          0.0|
|  5|       c|          1.0|
+---+--------+-------------+
```

8.2.8 labelConverter

```
from pyspark.ml.feature import IndexToString, StringIndexer

df = spark.createDataFrame(
    [(0, "Yes"), (1, "Yes"), (2, "Yes"), (3, "No"), (4, "No"), (5, "No")],
    ["id", "label"])

indexer = StringIndexer(inputCol="label", outputCol="labelIndex")
model = indexer.fit(df)
indexed = model.transform(df)

print("Transformed string column '%s' to indexed column '%s'"
      % (indexer.getInputCol(), indexer.getOutputCol()))
indexed.show()

print("StringIndexer will store labels in output column metadata\n")

converter = IndexToString(inputCol="labelIndex", outputCol="originalLabel")
converted = converter.transform(indexed)

print("Transformed indexed column '%s' back to original string column '%s'␣
↪using "
      "labels in metadata" % (converter.getInputCol(), converter.
↪getOutputCol()))
converted.select("id", "labelIndex", "originalLabel").show()
```

```
Transformed string column 'label' to indexed column 'labelIndex'
+---+-----+----------+
| id|label|labelIndex|
+---+-----+----------+
|  0|  Yes|       1.0|
|  1|  Yes|       1.0|
|  2|  Yes|       1.0|
|  3|   No|       0.0|
|  4|   No|       0.0|
|  5|   No|       0.0|
+---+-----+----------+
```

(continues on next page)

(continued from previous page)

```
StringIndexer will store labels in output column metadata

Transformed indexed column 'labelIndex' back to original string column
↪'originalLabel' using labels in metadata
+---+----------+-------------+
| id|labelIndex|originalLabel|
+---+----------+-------------+
|  0|       1.0|          Yes|
|  1|       1.0|          Yes|
|  2|       1.0|          Yes|
|  3|       0.0|           No|
|  4|       0.0|           No|
|  5|       0.0|           No|
+---+----------+-------------+
```

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import IndexToString, StringIndexer

df = spark.createDataFrame(
    [(0, "Yes"), (1, "Yes"), (2, "Yes"), (3, "No"), (4, "No"), (5, "No")],
    ["id", "label"])

indexer = StringIndexer(inputCol="label", outputCol="labelIndex")
converter = IndexToString(inputCol="labelIndex", outputCol="originalLabel")

pipeline = Pipeline(stages=[indexer, converter])


model = pipeline.fit(df)
result = model.transform(df)

result.show()
```

```
+---+-----+----------+-------------+
| id|label|labelIndex|originalLabel|
+---+-----+----------+-------------+
|  0|  Yes|       1.0|          Yes|
|  1|  Yes|       1.0|          Yes|
|  2|  Yes|       1.0|          Yes|
|  3|   No|       0.0|           No|
|  4|   No|       0.0|           No|
|  5|   No|       0.0|           No|
+---+-----+----------+-------------+
```

8.2.9 VectorIndexer

```
from pyspark.ml import Pipeline
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.ml.feature import RFormula

df = spark.createDataFrame([
    (0, 2.2, True, "1", "foo", 'CA'),
    (1, 3.3, False, "2", "bar", 'US'),
    (0, 4.4, False, "3", "baz", 'CHN'),
    (1, 5.5, False, "4", "foo", 'AUS')
], ['label',"real", "bool", "stringNum", "string","country"])

formula = RFormula(
    formula="label ~ real + bool + stringNum + string + country",
    featuresCol="features",
    labelCol="label")

# Automatically identify categorical features, and index them.
# We specify maxCategories so features with > 4 distinct values
# are treated as continuous.
featureIndexer = VectorIndexer(inputCol="features", \
                               outputCol="indexedFeatures",\
                               maxCategories=2)

pipeline = Pipeline(stages=[formula, featureIndexer])

model = pipeline.fit(df)
result = model.transform(df)

result.show()
```

```
+-----+----+-----+---------+------+-------+-------------------+-------------
↪------+
|label|real| bool|stringNum|string|country|           features|
↪indexedFeatures|
+-----+----+-----+---------+------+-------+-------------------+-------------
↪------+
|    0| 2.2| true|        1|   foo|     CA|(10,[0,1,5,7],[2....|(10,[0,1,5,7],
↪[2....|
|    1| 3.3|false|        2|   bar|     US|(10,[0,3,8],[3.3,...|(10,[0,3,8],
↪[3.3,...|
|    0| 4.4|false|        3|   baz|    CHN|(10,[0,4,6,9],[4....|(10,[0,4,6,9],
↪[4....|
|    1| 5.5|false|        4|   foo|    AUS|(10,[0,2,5],[5.5,...|(10,[0,2,5],
↪[5.5,...|
+-----+----+-----+---------+------+-------+-------------------+-------------
↪------+
```

8.2.10 VectorAssembler

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

dataset = spark.createDataFrame(
    [(0, 18, 1.0, Vectors.dense([0.0, 10.0, 0.5]), 1.0)],
    ["id", "hour", "mobile", "userFeatures", "clicked"])

assembler = VectorAssembler(
    inputCols=["hour", "mobile", "userFeatures"],
    outputCol="features")

output = assembler.transform(dataset)
print("Assembled columns 'hour', 'mobile', 'userFeatures' to vector column
↪'features'")
output.select("features", "clicked").show(truncate=False)
```

```
Assembled columns 'hour', 'mobile', 'userFeatures' to vector column 'features'
+-----------------------+-------+
|features               |clicked|
+-----------------------+-------+
|[18.0,1.0,0.0,10.0,0.5]|1.0    |
+-----------------------+-------+
```

8.2.11 OneHotEncoder

This is the note I wrote for one of my readers for explaining the OneHotEncoder. I would like to share it at here:

Import and creating SparkSession

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

```
df = spark.createDataFrame([
    (0, "a"),
    (1, "b"),
    (2, "c"),
    (3, "a"),
    (4, "a"),
    (5, "c")
], ["id", "category"])
df.show()
```

```
+---+--------+
| id|category|
+---+--------+
|  0|       a|
|  1|       b|
|  2|       c|
|  3|       a|
|  4|       a|
|  5|       c|
+---+--------+
```

OneHotEncoder

Encoder

```
from pyspark.ml.feature import OneHotEncoder, StringIndexer


stringIndexer = StringIndexer(inputCol="category", outputCol="categoryIndex")
model = stringIndexer.fit(df)
indexed = model.transform(df)

# default setting: dropLast=True
encoder = OneHotEncoder(inputCol="categoryIndex", outputCol="categoryVec",
↪dropLast=False)
encoded = encoder.transform(indexed)
encoded.show()
```

```
+---+--------+-------------+-------------+
| id|category|categoryIndex|  categoryVec|
+---+--------+-------------+-------------+
|  0|       a|          0.0|(3,[0],[1.0])|
|  1|       b|          2.0|(3,[2],[1.0])|
|  2|       c|          1.0|(3,[1],[1.0])|
|  3|       a|          0.0|(3,[0],[1.0])|
|  4|       a|          0.0|(3,[0],[1.0])|
|  5|       c|          1.0|(3,[1],[1.0])|
+---+--------+-------------+-------------+
```

Note: The default setting of `OneHotEncoder` is: dropLast=True

```
# default setting: dropLast=True
encoder = OneHotEncoder(inputCol="categoryIndex", outputCol=
↪"categoryVec")
encoded = encoder.transform(indexed)
encoded.show()
```

```
+---+--------+-------------+-------------+
| id|category|categoryIndex|  categoryVec|
+---+--------+-------------+-------------+
|  0|       a|          0.0|(2,[0],[1.0])|
|  1|       b|          2.0|    (2,[],[])|
|  2|       c|          1.0|(2,[1],[1.0])|
|  3|       a|          0.0|(2,[0],[1.0])|
|  4|       a|          0.0|(2,[0],[1.0])|
|  5|       c|          1.0|(2,[1],[1.0])|
+---+--------+-------------+-------------+
```

Vector Assembler

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler
categoricalCols = ['category']

indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
                for c in categoricalCols ]
# default setting: dropLast=True
encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                outputCol="{0}_encoded".format(indexer.getOutputCol()),
 ↪dropLast=False)
                for indexer in indexers ]
assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in␣
 ↪encoders]
                            , outputCol="features")
pipeline = Pipeline(stages=indexers + encoders + [assembler])

model=pipeline.fit(df)
data = model.transform(df)
```

```
data.show()
+---+--------+---------------+-----------------------+-------------+
| id|category|category_indexed|category_indexed_encoded|     features|
+---+--------+---------------+-----------------------+-------------+
|  0|       a|            0.0|          (3,[0],[1.0])|[1.0,0.0,0.0]|
|  1|       b|            2.0|          (3,[2],[1.0])|[0.0,0.0,1.0]|
|  2|       c|            1.0|          (3,[1],[1.0])|[0.0,1.0,0.0]|
|  3|       a|            0.0|          (3,[0],[1.0])|[1.0,0.0,0.0]|
|  4|       a|            0.0|          (3,[0],[1.0])|[1.0,0.0,0.0]|
|  5|       c|            1.0|          (3,[1],[1.0])|[0.0,1.0,0.0]|
+---+--------+---------------+-----------------------+-------------+
```

Application: Get Dummy Variable

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol,
↪dropLast=False):

    '''
    Get dummy variables and concat with continuous variables for ml modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols:  the name list of the numerical data
    :param labelCol:  the name of label column
    :param dropLast:  the flag of drop last column
    :return: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com

    >>> df = spark.createDataFrame([
                    (0, "a"),
                    (1, "b"),
                    (2, "c"),
                    (3, "a"),
                    (4, "a"),
                    (5, "c")
                ], ["id", "category"])

    >>> indexCol = 'id'
    >>> categoricalCols = ['category']
    >>> continuousCols = []
    >>> labelCol = []

    >>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol)
    >>> mat.show()

    >>>
        +---+-------------+
        | id|     features|
        +---+-------------+
        |  0|[1.0,0.0,0.0]|
        |  1|[0.0,0.0,1.0]|
        |  2|[0.0,1.0,0.0]|
        |  3|[1.0,0.0,0.0]|
        |  4|[1.0,0.0,0.0]|
        |  5|[0.0,1.0,0.0]|
        +---+-------------+
    '''

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder,␣
↪VectorAssembler
    from pyspark.sql.functions import col
```

(continues on next page)

(continued from previous page)

```
    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.getOutputCol()),
↪dropLast=dropLast)
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder␣
↪in encoders]
                                + continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    if indexCol and labelCol:
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
        return data.select(indexCol,'features','label')
    elif not indexCol and labelCol:
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
        return data.select('features','label')
    elif indexCol and not labelCol:
        # for unsupervised learning
        return data.select(indexCol,'features')
    elif not indexCol and not labelCol:
        # for unsupervised learning
        return data.select('features')
```

Unsupervised scenario

```
df = spark.createDataFrame([
    (0, "a"),
    (1, "b"),
    (2, "c"),
    (3, "a"),
    (4, "a"),
    (5, "c")
], ["id", "category"])
df.show()

indexCol = 'id'
categoricalCols = ['category']
continuousCols = []
labelCol = []
```

(continues on next page)

(continued from previous page)

```
mat = get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol)
```

```
mat.show()

+---+-------------+
| id|     features|
+---+-------------+
|  0|[1.0,0.0,0.0]|
|  1|[0.0,0.0,1.0]|
|  2|[0.0,1.0,0.0]|
|  3|[1.0,0.0,0.0]|
|  4|[1.0,0.0,0.0]|
|  5|[0.0,1.0,0.0]|
+---+-------------+
```

Supervised scenario

```
df = spark.read.csv(path='bank.csv',
                    sep=',',encoding='UTF-8',comment=None,
                    header=True,inferSchema=True)

indexCol = []
catCols = ['job','marital','education','default',
           'housing','loan','contact','poutcome']

contCols = ['balance', 'duration','campaign','pdays','previous']
labelCol = 'y'

data = get_dummy(df,indexCol,catCols,contCols,labelCol,dropLast=False)
data.show(5)
```

```
+--------------------+-----+
|            features|label|
+--------------------+-----+
|(37,[8,12,17,19,2...|   no|
|(37,[4,12,15,19,2...|   no|
|(37,[0,13,16,19,2...|   no|
|(37,[0,12,16,19,2...|   no|
|(37,[1,12,15,19,2...|   no|
+--------------------+-----+
only showing top 5 rows
```

The Jupyter Notebook can be found on Colab: OneHotEncoder .

8.2.12 Scaler

```
from pyspark.ml.feature import Normalizer, StandardScaler, MinMaxScaler,␣
↪MaxAbsScaler

scaler_type = 'Normal'
if scaler_type=='Normal':
    scaler = Normalizer(inputCol="features", outputCol="scaledFeatures", p=1.
↪0)
elif scaler_type=='Standard':
    scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures",
                            withStd=True, withMean=False)
elif scaler_type=='MinMaxScaler':
    scaler = MinMaxScaler(inputCol="features", outputCol="scaledFeatures")
elif scaler_type=='MaxAbsScaler':
    scaler = MaxAbsScaler(inputCol="features", outputCol="scaledFeatures")
```

```
from pyspark.ml import Pipeline
from pyspark.ml.linalg import Vectors

df = spark.createDataFrame([
    (0, Vectors.dense([1.0, 0.5, -1.0]),),
    (1, Vectors.dense([2.0, 1.0, 1.0]),),
    (2, Vectors.dense([4.0, 10.0, 2.0]),)
], ["id", "features"])
df.show()


pipeline = Pipeline(stages=[scaler])

model  =pipeline.fit(df)
data = model.transform(df)
data.show()
```

```
+---+--------------+
| id|      features|
+---+--------------+
|  0|[1.0,0.5,-1.0]|
|  1| [2.0,1.0,1.0]|
|  2|[4.0,10.0,2.0]|
+---+--------------+

+---+--------------+------------------+
| id|      features|    scaledFeatures|
+---+--------------+------------------+
|  0|[1.0,0.5,-1.0]|    [0.4,0.2,-0.4]|
|  1| [2.0,1.0,1.0]|   [0.5,0.25,0.25]|
|  2|[4.0,10.0,2.0]|[0.25,0.625,0.125]|
+---+--------------+------------------+
```

Normalizer

```
from pyspark.ml.feature import Normalizer
from pyspark.ml.linalg import Vectors

dataFrame = spark.createDataFrame([
    (0, Vectors.dense([1.0, 0.5, -1.0]),),
    (1, Vectors.dense([2.0, 1.0, 1.0]),),
    (2, Vectors.dense([4.0, 10.0, 2.0]),)
], ["id", "features"])

# Normalize each Vector using $L^1$ norm.
normalizer = Normalizer(inputCol="features", outputCol="normFeatures", p=1.0)
l1NormData = normalizer.transform(dataFrame)
print("Normalized using L^1 norm")
l1NormData.show()

# Normalize each Vector using $L^\infty$ norm.
lInfNormData = normalizer.transform(dataFrame, {normalizer.p: float("inf")})
print("Normalized using L^inf norm")
lInfNormData.show()
```

```
Normalized using L^1 norm
+---+--------------+------------------+
| id|      features|      normFeatures|
+---+--------------+------------------+
|  0|[1.0,0.5,-1.0]|    [0.4,0.2,-0.4]|
|  1| [2.0,1.0,1.0]|   [0.5,0.25,0.25]|
|  2|[4.0,10.0,2.0]|[0.25,0.625,0.125]|
+---+--------------+------------------+

Normalized using L^inf norm
+---+--------------+--------------+
| id|      features|  normFeatures|
+---+--------------+--------------+
|  0|[1.0,0.5,-1.0]|[1.0,0.5,-1.0]|
|  1| [2.0,1.0,1.0]| [1.0,0.5,0.5]|
|  2|[4.0,10.0,2.0]| [0.4,1.0,0.2]|
+---+--------------+--------------+
```

StandardScaler

```
from pyspark.ml.feature import Normalizer, StandardScaler, MinMaxScaler,␣
↪MaxAbsScaler

from pyspark.ml.linalg import Vectors

dataFrame = spark.createDataFrame([
    (0, Vectors.dense([1.0, 0.5, -1.0]),),
    (1, Vectors.dense([2.0, 1.0, 1.0]),),
```

(continues on next page)

(continued from previous page)

```
    (2, Vectors.dense([4.0, 10.0, 2.0]),)
], ["id", "features"])

scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures",
                        withStd=True, withMean=False)
scaledData = scaler.fit((dataFrame)).transform(dataFrame)
scaledData.show(truncate=False)
```

```
+---+--------------+-------------------------------------------------------------
↪--+
|id |features      |scaledFeatures
↪   |
+---+--------------+-------------------------------------------------------------
↪--+
|0  |[1.0,0.5,-1.0]|[0.6546536707079772,0.09352195295828244,-0.
↪6546536707079771]|
|1  |[2.0,1.0,1.0] |[1.3093073414159544,0.1870439059165649,0.
↪6546536707079771]  |
|2  |[4.0,10.0,2.0]|[2.618614682831909,1.870439059165649,1.3093073414159542]
↪   |
+---+--------------+-------------------------------------------------------------
↪--+
```

MinMaxScaler

```
from pyspark.ml.feature import Normalizer, StandardScaler, MinMaxScaler,␣
↪MaxAbsScaler

from pyspark.ml.linalg import Vectors

dataFrame = spark.createDataFrame([
    (0, Vectors.dense([1.0, 0.5, -1.0]),),
    (1, Vectors.dense([2.0, 1.0, 1.0]),),
    (2, Vectors.dense([4.0, 10.0, 2.0]),)
], ["id", "features"])

scaler = MinMaxScaler(inputCol="features", outputCol="scaledFeatures")
scaledData = scaler.fit((dataFrame)).transform(dataFrame)
scaledData.show(truncate=False)
```

```
+---+--------------+-------------------------------------------------------------
↪-+
|id |features      |scaledFeatures
↪  |
+---+--------------+-------------------------------------------------------------
↪-+
|0  |[1.0,0.5,-1.0]|[0.0,0.0,0.0]
↪  |
|1  |[2.0,1.0,1.0] |[0.3333333333333333,0.05263157894736842,0.
↪6666666666666666]|
```

(continues on next page)

(continued from previous page)

```
|2  |[4.0,10.0,2.0]|[1.0,1.0,1.0]                                                  ␣
↪ |
+---+--------------+-----------------------------------------------------------
↪-+
```

MaxAbsScaler

```
from pyspark.ml.feature import Normalizer, StandardScaler, MinMaxScaler,␣
↪MaxAbsScaler

from pyspark.ml.linalg import Vectors

dataFrame = spark.createDataFrame([
    (0, Vectors.dense([1.0, 0.5, -1.0]),),
    (1, Vectors.dense([2.0, 1.0, 1.0]),),
    (2, Vectors.dense([4.0, 10.0, 2.0]),)
], ["id", "features"])

scaler = MaxAbsScaler(inputCol="features", outputCol="scaledFeatures")
scaledData = scaler.fit((dataFrame)).transform(dataFrame)
scaledData.show(truncate=False)
```

```
+---+--------------+----------------+
|id |features      |scaledFeatures  |
+---+--------------+----------------+
|0  |[1.0,0.5,-1.0]|[0.25,0.05,-0.5]|
|1  |[2.0,1.0,1.0] |[0.5,0.1,0.5]   |
|2  |[4.0,10.0,2.0]|[1.0,1.0,1.0]   |
+---+--------------+----------------+
```

8.2.13 PCA

```
from pyspark.ml.feature import PCA
from pyspark.ml.linalg import Vectors

data = [(Vectors.sparse(5, [(1, 1.0), (3, 7.0)]),),
        (Vectors.dense([2.0, 0.0, 3.0, 4.0, 5.0]),),
        (Vectors.dense([4.0, 0.0, 0.0, 6.0, 7.0]),)]
df = spark.createDataFrame(data, ["features"])

pca = PCA(k=3, inputCol="features", outputCol="pcaFeatures")
model = pca.fit(df)

result = model.transform(df).select("pcaFeatures")
result.show(truncate=False)
```

```
+-------------------------------------------------------------+
|pcaFeatures                                                  |
+-------------------------------------------------------------+
|[1.6485728230883807,-4.013282700516296,-5.524543751369388]  |
|[-4.645104331781534,-1.1167972663619026,-5.524543751369387]|
|[-6.428880535676489,-5.337951427775355,-5.524543751369389]  |
+-------------------------------------------------------------+
```

8.2.14 DCT

```
from pyspark.ml.feature import DCT
from pyspark.ml.linalg import Vectors

df = spark.createDataFrame([
    (Vectors.dense([0.0, 1.0, -2.0, 3.0]),),
    (Vectors.dense([-1.0, 2.0, 4.0, -7.0]),),
    (Vectors.dense([14.0, -2.0, -5.0, 1.0]),)], ["features"])

dct = DCT(inverse=False, inputCol="features", outputCol="featuresDCT")

dctDf = dct.transform(df)

dctDf.select("featuresDCT").show(truncate=False)
```

```
+----------------------------------------------------------------+
|featuresDCT                                                     |
+----------------------------------------------------------------+
|[1.0,-1.1480502970952693,2.0000000000000004,-2.7716385975338604]|
|[-1.0,3.378492794482933,-7.000000000000001,2.9301512653149677]  |
|[4.0,9.304453421915744,11.000000000000002,1.5579302036357163]   |
+----------------------------------------------------------------+
```

8.3 Feature Selection

8.3.1 LASSO

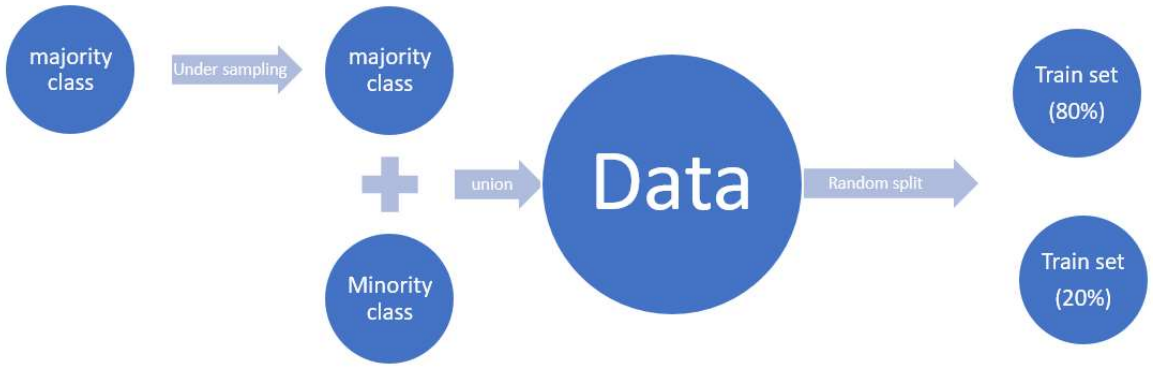
Variable selection and the removal of correlated variables. The Ridge method shrinks the coefficients of correlated variables while the LASSO method picks one variable and discards the others. The elastic net penalty is a mixture of these two; if variables are correlated in groups then $\alpha = 0.5$ tends to select the groups as in or out. If is close to 1, the elastic net performs much like the LASSO method and removes any degeneracies and wild behavior caused by extreme correlations.

8.3.2 RandomForest

AutoFeatures library based on RandomForest is coming soon. . . . . .

8.4 Unbalanced data: Undersampling

Since we use PySpark to deal with the big data, Undersampling for Unbalanced Classification is a useful method to deal with the Unbalanced data. Undersampling is a popular technique for unbalanced datasets to reduce the skew in class distributions. However, it is well-known that undersampling one class modifies the priors of the training set and consequently biases the posterior probabilities of a classifier. After you applied the Undersampling, you need to recalibrate the Probability Calibrating Probability with Undersampling for Unbalanced Classification.




```
df = spark.createDataFrame([
    (0, "Yes"),
    (1, "Yes"),
    (2, "Yes"),
    (3, "Yes"),
    (4, "No"),
    (5, "No")
], ["id", "label"])
df.show()
```

```
+---+-----+
| id|label|
+---+-----+
|  0|  Yes|
|  1|  Yes|
|  2|  Yes|
|  3|  Yes|
|  4|   No|
|  5|   No|
+---+-----+
```

8.4.1 Calculate undersampling Ratio

```
import math
def round_up(n, decimals=0):
    multiplier = 10 ** decimals
    return math.ceil(n * multiplier) / multiplier

  # drop missing value rows
df = df.dropna()
# under-sampling majority set
label_Y = df.filter(df.label=='Yes')
label_N = df.filter(df.label=='No')
sampleRatio = round_up(label_N.count() / df.count(),2)
```

8.4.2 Undersampling

```
label_Y_sample = label_Y.sample(False, sampleRatio)
# union minority set and the under-sampling majority set
data = label_N.unionAll(label_Y_sample)
data.show()
```

```
+---+-----+
| id|label|
+---+-----+
|  4|   No|
|  5|   No|
|  1|  Yes|
|  2|  Yes|
+---+-----+
```

8.4.3 Recalibrating Probability

Undersampling is a popular technique for unbalanced datasets to reduce the skew in class distributions. However, it is well-known that undersampling one class modifies the priors of the training set and consequently biases the posterior probabilities of a classifier Calibrating Probability with Undersampling for Unbalanced Classification.

```
predication.withColumn('adj_probability',sampleRatio*F.col('probability')/
↪((sampleRatio-1)*F.col('probability')+1))
```

CHAPTER

NINE

REGRESSION

Chinese proverb

A journey of a thousand miles begins with a single step. – old Chinese proverb

In statistical modeling, regression analysis focuses on investigating the relationship between a dependent variable and one or more independent variables. Wikipedia Regression analysis

In data mining, Regression is a model to represent the relationship between the value of lable (or target, it is numerical variable) and on one or more features (or predictors they can be numerical and categorical variables).

9.1 Linear Regression

9.1.1 Introduction

Given that a data set $\{ x_{i1}, \ldots, x_{in}, y_i\}_{i=1}^{m}$ which contains n features (variables) and m samples (data points), in simple linear regression model for modeling m data points with j independent variables: x_{ij}, the formula is given by:

$$y_i = \beta_0 + \beta_j x_{ij}, \text{where}, i = 1, \cdots m, j = 1, \cdots n.$$

In matrix notation, the data set is written as $\mathbf{X} = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n]$ with $\boldsymbol{x}_j = \{x_{ij}\}_{i=1}^{m}$, $\boldsymbol{y} = \{y_i\}_{i=1}^{m}$ (see Fig. *Feature matrix and label*) and $\boldsymbol{\beta}^\top = \{\beta_j\}_{j=1}^{n}$. Then the matrix format equation is written as

$$\boldsymbol{y} = \mathbf{X}\boldsymbol{\beta}. \tag{9.1}$$

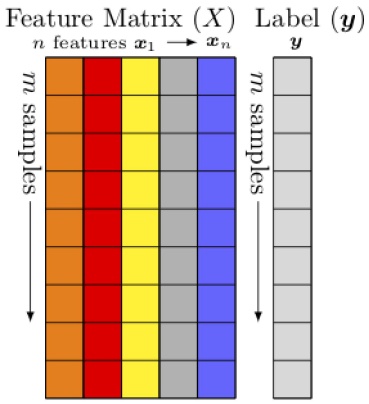


Fig. 1: Feature matrix and label

9.1.2 How to solve it?

1. Direct Methods (For more information please refer to my Prelim Notes for Numerical Analysis)
    - For squared or rectangular matrices
        - Singular Value Decomposition
        - Gram-Schmidt orthogonalization
        - QR Decomposition
    - For squared matrices
        - LU Decomposition
        - Cholesky Decomposition
        - Regular Splittings
2. Iterative Methods
    - Stationary cases iterative method
        - Jacobi Method
        - Gauss-Seidel Method
        - Richardson Method
        - Successive Over Relaxation (SOR) Method
    - Dynamic cases iterative method
        - Chebyshev iterative Method
        - Minimal residuals Method
        - Minimal correction iterative method
        - Steepest Descent Method

- Conjugate Gradients Method

9.1.3 Ordinary Least Squares

In mathematics, (9.1) is a overdetermined system. The method of ordinary least squares can be used to find an approximate solution to overdetermined systems. For the system overdetermined system (9.1), the least squares formula is obtained from the problem

$$\min_{\boldsymbol{\beta}} ||\mathbf{X}\boldsymbol{\beta} - \boldsymbol{y}||, \tag{9.2}$$

the solution of which can be written with the normal equations:

$$\boldsymbol{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\boldsymbol{y} \tag{9.3}$$

where T indicates a matrix transpose, provided $(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}$ exists (that is, provided $\mathbf{X}$ has full column rank).

Note: Actually, (9.3) can be derivated by the following way: multiply $\mathbf{X}^T$ on side of (9.1) and then multiply $(\mathbf{X}^T\mathbf{X})^{-1}$ on both side of the former result. You may also apply the `Extreme Value Theorem` to (9.2) and find the solution (9.3).

9.1.4 Gradient Descent

Let's use the following hypothesis:

$$h_{\boldsymbol{\beta}} = \beta_0 + \beta_j \boldsymbol{x}_j, \text{where}, j = 1, \cdots n.$$

Then, solving (9.2) is equivalent to minimize the following `cost fucntion` :

9.1.5 Cost Function

$$J(\boldsymbol{\beta}) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_{\boldsymbol{\beta}}(x^{(i)}) - y^{(i)})\right)^2 \tag{9.4}$$

Note: The reason why we prefer to solve (9.4) rather than (9.2) is because (9.4) is convex and it has some nice properties, such as it's uniquely solvable and energy stable for small enough learning rate. the interested reader who has great interest in non-convex cost function (energy) case. is referred to [Feng2016PSD] for more details.

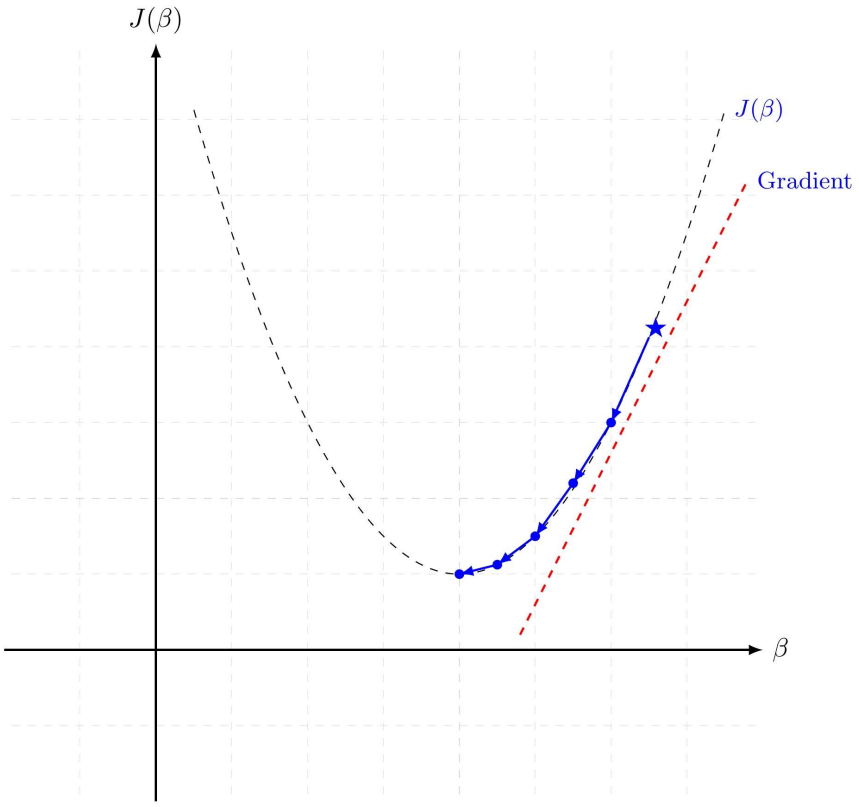


Fig. 2: Gradient Descent in 1D

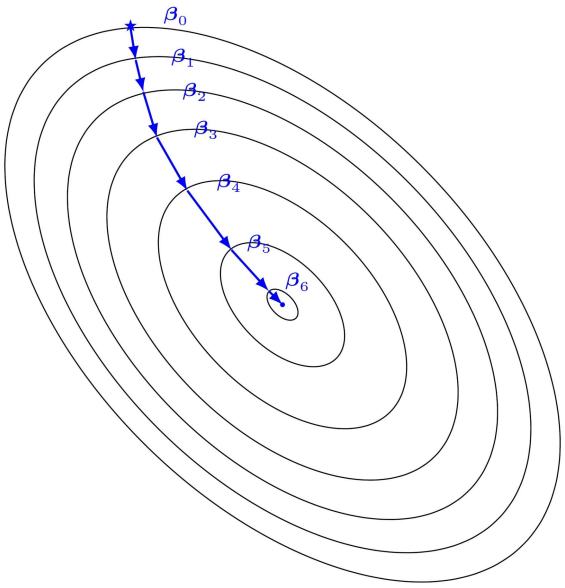


Fig. 3: Gradient Descent in 2D

9.1.6 Batch Gradient Descent

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. It searchs with the direction of the steepest desscent which is defined by the `negative of the gradient` (see Fig. *Gradient Descent in 1D* and *Gradient Descent in 2D* for 1D and 2D, respectively) and with learning rate (search step) α.

9.1.7 Stochastic Dradient Descent

9.1.8 Mini-batch Gradient Descent

9.1.9 Demo

- The Jupyter notebook can be download from Linear Regression which was implemented without using Pipeline.
- The Jupyter notebook can be download from Linear Regression with Pipeline which was implemented with using Pipeline.
- I will only present the code with pipeline style in the following.
- For more details about the parameters, please visit Linear Regression API .

1. Set up spark context and SparkSession

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark regression example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

2. Load dataset

```
df = spark.read.format('com.databricks.spark.csv').\
                        options(header='true', \
                        inferschema='true').\
            load("../data/Advertising.csv",header=True);
```

check the data set

```
df.show(5,True)
df.printSchema()
```

Then you will get

```
+-----+-----+---------+-----+
|   TV|Radio|Newspaper|Sales|
+-----+-----+---------+-----+
|230.1| 37.8|     69.2| 22.1|
```

(continues on next page)

(continued from previous page)

```
| 44.5| 39.3|      45.1| 10.4|
| 17.2| 45.9|      69.3|  9.3|
|151.5| 41.3|      58.5| 18.5|
|180.8| 10.8|      58.4| 12.9|
+-----+-----+---------+-----+
only showing top 5 rows

root
 |-- TV: double (nullable = true)
 |-- Radio: double (nullable = true)
 |-- Newspaper: double (nullable = true)
 |-- Sales: double (nullable = true)
```

You can also get the Statistical results from the data frame (Unfortunately, it only works for numerical).

```
df.describe().show()
```

Then you will get

```
+-------+-----------------+-----------------+------------------+------------
↪-----+
|summary|               TV|            Radio|         Newspaper|
↪Sales|
+-------+-----------------+-----------------+------------------+------------
↪-----+
|  count|              200|              200|               200|
↪  200|
|   mean|         147.0425|23.264000000000024|30.553999999999995|14.
↪022500000000003|
| stddev|85.85423631490805|14.846809176168728| 21.77862083852283| 5.
↪217456565710477|
|    min|              0.7|              0.0|               0.3|
↪  1.6|
|    max|            296.4|             49.6|             114.0|
↪ 27.0|
+-------+-----------------+-----------------+------------------+------------
↪-----+
```

Fig. 4: Sales distribution

3. Convert the data to dense vector (**features** and **label**)

```
from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

# I provide two ways to build the features and labels

# method 1 (good for small feature):
#def transData(row):
#    return Row(label=row["Sales"],
#               features=Vectors.dense([row["TV"],
#                                       row["Radio"],
#                                       row["Newspaper"]]))

# Method 2 (good for large features):
def transData(data):
return data.rdd.map(lambda r: [Vectors.dense(r[:-1]),r[-1]]).toDF(['features',
↪'label'])
```

Note:

You are strongly encouraged to try my `get_dummy` function for dealing with the categorical data in comple dataset.

Supervised learning version:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,
↪labelCol):
```

(continues on next page)

(continued from previous page)

```
    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer,␣
↪OneHotEncoder, VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
↪indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
↪getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
↪getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol=
↪"features")

    pipeline = Pipeline(stages=indexers + encoders +␣
↪[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label',col(labelCol))

    if indexCol:
        return data.select(indexCol,'features','label')
    else:
        return data.select('features','label')
```

Unsupervised learning version:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols):
    '''
    Get dummy variables and concat with continuous variables␣
↪for unsupervised learning.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical␣
↪data
    :param continuousCols:  the name list of the numerical␣
↪data
    :return k: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com
    '''
```

(continues on next page)

(continued from previous page)

```
    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
↪indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
↪getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
↪getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol=
↪"features")

    pipeline = Pipeline(stages=indexers + encoders +␣
↪[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    if indexCol:
        return data.select(indexCol,'features')
    else:
        return data.select('features')
```

Two in one:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol,
↪dropLast=False):

    '''
    Get dummy variables and concat with continuous variables for ml␣
↪modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols:  the name list of the numerical data
    :param labelCol:  the name of label column
    :param dropLast:  the flag of drop last column
    :return: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com

    >>> df = spark.createDataFrame([
                    (0, "a"),
                    (1, "b"),
                    (2, "c"),
                    (3, "a"),
                    (4, "a"),
                    (5, "c")
```

(continues on next page)

(continued from previous page)

```
            ], ["id", "category"])

    >>> indexCol = 'id'
    >>> categoricalCols = ['category']
    >>> continuousCols = []
    >>> labelCol = []

    >>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,
↪labelCol)
    >>> mat.show()

    >>>
        +---+-------------+
        | id|     features|
        +---+-------------+
        |  0|[1.0,0.0,0.0]|
        |  1|[0.0,0.0,1.0]|
        |  2|[0.0,1.0,0.0]|
        |  3|[1.0,0.0,0.0]|
        |  4|[1.0,0.0,0.0]|
        |  5|[0.0,1.0,0.0]|
        +---+-------------+
    '''

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder,␣
↪VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".
↪format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()),dropLast=dropLast)
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol()␣
↪for encoder in encoders]
                                + continuousCols, outputCol="features
↪")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    if indexCol and labelCol:
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
```

(continues on next page)

(continued from previous page)

```
        return data.select(indexCol,'features','label')
    elif not indexCol and labelCol:
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
        return data.select('features','label')
    elif indexCol and not labelCol:
        # for unsupervised learning
        return data.select(indexCol,'features')
    elif not indexCol and not labelCol:
        # for unsupervised learning
        return data.select('features')
```

4. Transform the dataset to DataFrame

```
transformed= transData(df)
transformed.show(5)
```

```
+-----------------+-----+
|         features|label|
+-----------------+-----+
|[230.1,37.8,69.2]| 22.1|
| [44.5,39.3,45.1]| 10.4|
| [17.2,45.9,69.3]|  9.3|
|[151.5,41.3,58.5]| 18.5|
|[180.8,10.8,58.4]| 12.9|
+-----------------+-----+
only showing top 5 rows
```

Note: You will find out that all of the supervised machine learning algorithms in Spark are based on the **features** and **label** (unsupervised machine learning algorithms in Spark are based on the **features**). That is to say, you can play with all of the machine learning algorithms in Spark when you get ready the **features** and **label** in pipeline architecture.

5. Deal With Categorical Variables

```
from pyspark.ml import Pipeline
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

# Automatically identify categorical features, and index them.
# We specify maxCategories so features with > 4 distinct values are treated␣
↪as continuous.

featureIndexer = VectorIndexer(inputCol="features", \
                               outputCol="indexedFeatures",\
                               maxCategories=4).fit(transformed)
```

(continues on next page)

(continued from previous page)

```
data = featureIndexer.transform(transformed)
```

Now you check your dataset with

```
data.show(5,True)
```

you will get

```
+-----------------+-----+-----------------+
|         features|label|  indexedFeatures|
+-----------------+-----+-----------------+
|[230.1,37.8,69.2]| 22.1|[230.1,37.8,69.2]|
| [44.5,39.3,45.1]| 10.4| [44.5,39.3,45.1]|
| [17.2,45.9,69.3]|  9.3| [17.2,45.9,69.3]|
|[151.5,41.3,58.5]| 18.5|[151.5,41.3,58.5]|
|[180.8,10.8,58.4]| 12.9|[180.8,10.8,58.4]|
+-----------------+-----+-----------------+
only showing top 5 rows
```

6. Split the data into training and test sets (40% held out for testing)

```
# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = transformed.randomSplit([0.6, 0.4])
```

You can check your train and test data as follows (In my opinion, it is always to good to keep tracking your data during prototype pahse):

```
trainingData.show(5)
testData.show(5)
```

Then you will get

```
+---------------+-----+---------------+
|       features|label|indexedFeatures|
+---------------+-----+---------------+
| [4.1,11.6,5.7]|  3.2| [4.1,11.6,5.7]|
| [5.4,29.9,9.4]|  5.3| [5.4,29.9,9.4]|
|[7.3,28.1,41.4]|  5.5|[7.3,28.1,41.4]|
|[7.8,38.9,50.6]|  6.6|[7.8,38.9,50.6]|
|  [8.6,2.1,1.0]|  4.8|  [8.6,2.1,1.0]|
+---------------+-----+---------------+
only showing top 5 rows

+----------------+-----+----------------+
|        features|label| indexedFeatures|
+----------------+-----+----------------+
|  [0.7,39.6,8.7]|  1.6|  [0.7,39.6,8.7]|
|  [8.4,27.2,2.1]|  5.7|  [8.4,27.2,2.1]|
|[11.7,36.9,45.2]|  7.3|[11.7,36.9,45.2]|
|[13.2,15.9,49.6]|  5.6|[13.2,15.9,49.6]|
```

(continues on next page)

(continued from previous page)

```
|[16.9,43.7,89.4]|  8.7|[16.9,43.7,89.4]|
+----------------+-----+----------------+
only showing top 5 rows
```

7. Fit Ordinary Least Square Regression Model

For more details about the parameters, please visit Linear Regression API .

```
# Import LinearRegression class
from pyspark.ml.regression import LinearRegression

# Define LinearRegression algorithm
lr = LinearRegression()
```

8. Pipeline Architecture

```
# Chain indexer and tree in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, lr])

model = pipeline.fit(trainingData)
```

9. Summary of the Model

Spark has a poor summary function for data and model. I wrote a summary function which has similar format as **R** output for the linear regression in PySpark.

```
def modelsummary(model):
    import numpy as np
    print ("Note: the last rows are the information for Intercept")
    print ("##","------------------------------------------------")
    print ("##","  Estimate   |   Std.Error | t Values  |  P-value")
    coef = np.append(list(model.coefficients),model.intercept)
    Summary=model.summary

    for i in range(len(Summary.pValues)):
        print ("##",'{:10.6f}'.format(coef[i]),\
        '{:10.6f}'.format(Summary.coefficientStandardErrors[i]),\
        '{:8.3f}'.format(Summary.tValues[i]),\
        '{:10.6f}'.format(Summary.pValues[i]))

    print ("##",'---')
    print ("##","Mean squared error: % .6f" \
           % Summary.meanSquaredError, ", RMSE: % .6f" \
           % Summary.rootMeanSquaredError )
    print ("##","Multiple R-squared: %f" % Summary.r2, ", \
            Total iterations: %i"% Summary.totalIterations)
```

```
modelsummary(model.stages[-1])
```

You will get the following summary results:

```
Note: the last rows are the information for Intercept
('##', '-------------------------------------------------')
('##', '  Estimate   |   Std.Error | t Values  |  P-value')
('##', '  0.044186', '  0.001663', '  26.573', '  0.000000')
('##', '  0.206311', '  0.010846', '  19.022', '  0.000000')
('##', '  0.001963', '  0.007467', '   0.263', '  0.793113')
('##', '  2.596154', '  0.379550', '   6.840', '  0.000000')
('##', '---')
('##', 'Mean squared error:  2.588230', ', RMSE:  1.608798')
('##', 'Multiple R-squared: 0.911869', ',             Total iterations: 1')
```

10. Make predictions

```
# Make predictions.
predictions = model.transform(testData)
```

```
# Select example rows to display.
predictions.select("features","label","prediction").show(5)
```

```
+---------------+-----+------------------+
|       features|label|        prediction|
+---------------+-----+------------------+
|  [0.7,39.6,8.7]|  1.6| 10.81405928637388|
|  [8.4,27.2,2.1]|  5.7| 8.583086404079918|
|[11.7,36.9,45.2]|  7.3|10.814712818232422|
|[13.2,15.9,49.6]|  5.6| 6.557106943899219|
|[16.9,43.7,89.4]|  8.7|12.534151375058645|
+---------------+-----+------------------+
only showing top 5 rows
```

9. Evaluation

```
from pyspark.ml.evaluation import RegressionEvaluator
# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(labelCol="label",
                                predictionCol="prediction",
                                metricName="rmse")

rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

The final Root Mean Squared Error (RMSE) is as follows:

```
Root Mean Squared Error (RMSE) on test data = 1.63114
```

You can also check the R^2 value for the test data:

```
y_true = predictions.select("label").toPandas()
y_pred = predictions.select("prediction").toPandas()

import sklearn.metrics
```

(continues on next page)

(continued from previous page)

```
r2_score = sklearn.metrics.r2_score(y_true, y_pred)
print('r2_score: {0}'.format(r2_score))
```

Then you will get

```
r2_score: 0.854486655585
```

Warning: You should know most softwares are using different formula to calculate the R^2 value when no intercept is included in the model. You can get more information from the disscussion at StackExchange.

9.2 Generalized linear regression

9.2.1 Introduction

9.2.2 How to solve it?

9.2.3 Demo

- The Jupyter notebook can be download from Generalized Linear Regression.
- For more details about the parameters, please visit Generalized Linear Regression API .

1. Set up spark context and SparkSession

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark regression example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

2. Load dataset

```
df = spark.read.format('com.databricks.spark.csv').\
                        options(header='true', \
                        inferschema='true').\
            load("../data/Advertising.csv",header=True);
```

check the data set

```
df.show(5,True)
df.printSchema()
```

Then you will get

```
+-----+-----+---------+-----+
|   TV|Radio|Newspaper|Sales|
+-----+-----+---------+-----+
|230.1| 37.8|     69.2| 22.1|
| 44.5| 39.3|     45.1| 10.4|
| 17.2| 45.9|     69.3|  9.3|
|151.5| 41.3|     58.5| 18.5|
|180.8| 10.8|     58.4| 12.9|
+-----+-----+---------+-----+
only showing top 5 rows

root
 |-- TV: double (nullable = true)
 |-- Radio: double (nullable = true)
 |-- Newspaper: double (nullable = true)
 |-- Sales: double (nullable = true)
```

You can also get the Statistical resutls from the data frame (Unfortunately, it only works for numerical).

```
df.describe().show()
```

Then you will get

```
+-------+-----------------+------------------+------------------+------------
↪-----+
|summary|               TV|             Radio|         Newspaper|
↪Sales|
+-------+-----------------+------------------+------------------+------------
↪-----+
|  count|              200|               200|               200|
↪  200|
|   mean|         147.0425|23.264000000000024|30.553999999999995|14.
↪022500000000003|
| stddev|85.85423631490805|14.846809176168728| 21.77862083852283| 5.
↪217456565710477|
|    min|              0.7|               0.0|               0.3|
↪  1.6|
|    max|            296.4|              49.6|             114.0|
↪ 27.0|
+-------+-----------------+------------------+------------------+------------
↪-----+
```

3. Convert the data to dense vector (**features** and **label**)

Note:

You are strongly encouraged to try my `get_dummy` function for dealing with the categorical data in comple dataset.

Supervised learning version:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,
↪labelCol):

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer,␣
↪OneHotEncoder, VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
↪indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
↪getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
↪getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol=
↪"features")

    pipeline = Pipeline(stages=indexers + encoders +␣
↪[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label',col(labelCol))

    if indexCol:
        return data.select(indexCol,'features','label')
    else:
        return data.select('features','label')
```

Unsupervised learning version:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols):
    '''
    Get dummy variables and concat with continuous variables␣
↪for unsupervised learning.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical␣
↪data
    :param continuousCols:  the name list of the numerical␣
↪data
    :return k: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com
```

(continues on next page)

(continued from previous page)

```
    '''

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
↪indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
↪getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
↪getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol=
↪"features")

    pipeline = Pipeline(stages=indexers + encoders +␣
↪[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    if indexCol:
        return data.select(indexCol,'features')
    else:
        return data.select('features')
```

Two in one:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol,
↪dropLast=False):

    '''
    Get dummy variables and concat with continuous variables for ml␣
↪modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols:  the name list of the numerical data
    :param labelCol:  the name of label column
    :param dropLast:  the flag of drop last column
    :return: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com

    >>> df = spark.createDataFrame([
                    (0, "a"),
                    (1, "b"),
                    (2, "c"),
                    (3, "a"),
```

(continues on next page)

(continued from previous page)

```
                    (4, "a"),
                    (5, "c")
                ], ["id", "category"])

    >>> indexCol = 'id'
    >>> categoricalCols = ['category']
    >>> continuousCols = []
    >>> labelCol = []

    >>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,
↪labelCol)
    >>> mat.show()

    >>>
        +---+-------------+
        | id|     features|
        +---+-------------+
        |  0|[1.0,0.0,0.0]|
        |  1|[0.0,0.0,1.0]|
        |  2|[0.0,1.0,0.0]|
        |  3|[1.0,0.0,0.0]|
        |  4|[1.0,0.0,0.0]|
        |  5|[0.0,1.0,0.0]|
        +---+-------------+
    '''

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder,␣
↪VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".
↪format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()),dropLast=dropLast)
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol()␣
↪for encoder in encoders]
                                + continuousCols, outputCol="features
↪")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    if indexCol and labelCol:
```

(continues on next page)

(continued from previous page)

```
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
        return data.select(indexCol,'features','label')
    elif not indexCol and labelCol:
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
        return data.select('features','label')
    elif indexCol and not labelCol:
        # for unsupervised learning
        return data.select(indexCol,'features')
    elif not indexCol and not labelCol:
        # for unsupervised learning
        return data.select('features')
```

```
from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

# I provide two ways to build the features and labels

# method 1 (good for small feature):
#def transData(row):
#    return Row(label=row["Sales"],
#               features=Vectors.dense([row["TV"],
#                                       row["Radio"],
#                                       row["Newspaper"]]))

# Method 2 (good for large features):
def transData(data):
return data.rdd.map(lambda r: [Vectors.dense(r[:-1]),r[-1]]).toDF(['features',
↪'label'])
```

```
transformed= transData(df)
transformed.show(5)
```

```
+-----------------+-----+
|         features|label|
+-----------------+-----+
|[230.1,37.8,69.2]| 22.1|
| [44.5,39.3,45.1]| 10.4|
| [17.2,45.9,69.3]|  9.3|
|[151.5,41.3,58.5]| 18.5|
|[180.8,10.8,58.4]| 12.9|
+-----------------+-----+
only showing top 5 rows
```

Note: You will find out that all of the machine learning algorithms in Spark are based on the **features** and **label**. That is to say, you can play with all of the machine learning algorithms in Spark when you get ready

the **features** and **label**.

4. Convert the data to dense vector

```
# convert the data to dense vector
def transData(data):
    return data.rdd.map(lambda r: [r[-1], Vectors.dense(r[:-1])]).\
           toDF(['label','features'])

from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

data= transData(df)
data.show()
```

5. Deal with the Categorical variables

```
from pyspark.ml import Pipeline
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

# Automatically identify categorical features, and index them.
# We specify maxCategories so features with > 4
# distinct values are treated as continuous.

featureIndexer = VectorIndexer(inputCol="features", \
                               outputCol="indexedFeatures",\
                               maxCategories=4).fit(transformed)

data = featureIndexer.transform(transformed)
```

When you check you data at this point, you will get

```
+-----------------+-----+-----------------+
|         features|label|  indexedFeatures|
+-----------------+-----+-----------------+
|[230.1,37.8,69.2]| 22.1|[230.1,37.8,69.2]|
| [44.5,39.3,45.1]| 10.4| [44.5,39.3,45.1]|
| [17.2,45.9,69.3]|  9.3| [17.2,45.9,69.3]|
|[151.5,41.3,58.5]| 18.5|[151.5,41.3,58.5]|
|[180.8,10.8,58.4]| 12.9|[180.8,10.8,58.4]|
+-----------------+-----+-----------------+
only showing top 5 rows
```

6. Split the data into training and test sets (40% held out for testing)

```
# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = transformed.randomSplit([0.6, 0.4])
```

You can check your train and test data as follows (In my opinion, it is always to good to keep tracking your data during prototype phase):

```
trainingData.show(5)
testData.show(5)
```

Then you will get

```
+---------------+-----+----------------+
|       features|label| indexedFeatures|
+---------------+-----+----------------+
|  [5.4,29.9,9.4]|  5.3|  [5.4,29.9,9.4]|
| [7.8,38.9,50.6]|  6.6| [7.8,38.9,50.6]|
|  [8.4,27.2,2.1]|  5.7|  [8.4,27.2,2.1]|
| [8.7,48.9,75.0]|  7.2| [8.7,48.9,75.0]|
|[11.7,36.9,45.2]|  7.3|[11.7,36.9,45.2]|
+----------------+-----+----------------+
only showing top 5 rows

+---------------+-----+---------------+
|       features|label|indexedFeatures|
+---------------+-----+---------------+
| [0.7,39.6,8.7]|  1.6| [0.7,39.6,8.7]|
| [4.1,11.6,5.7]|  3.2| [4.1,11.6,5.7]|
|[7.3,28.1,41.4]|  5.5|[7.3,28.1,41.4]|
|  [8.6,2.1,1.0]|  4.8|  [8.6,2.1,1.0]|
|[17.2,4.1,31.6]|  5.9|[17.2,4.1,31.6]|
+---------------+-----+---------------+
only showing top 5 rows
```

7. Fit Generalized Linear Regression Model

```
# Import LinearRegression class
from pyspark.ml.regression import GeneralizedLinearRegression

# Define LinearRegression algorithm
glr = GeneralizedLinearRegression(family="gaussian", link="identity",\
                                  maxIter=10, regParam=0.3)
```

8. Pipeline Architecture

```
# Chain indexer and tree in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, glr])

model = pipeline.fit(trainingData)
```

9. Summary of the Model

Spark has a poor summary function for data and model. I wrote a summary function which has similar format as **R** output for the linear regression in PySpark.

```
def modelsummary(model):
    import numpy as np
    print ("Note: the last rows are the information for Intercept")
    print ("##","------------------------------------------------")
```

(continues on next page)

(continued from previous page)

```
    print ("##","  Estimate   |   Std.Error | t Values  |  P-value")
    coef = np.append(list(model.coefficients),model.intercept)
    Summary=model.summary

    for i in range(len(Summary.pValues)):
        print ("##",'{:10.6f}'.format(coef[i]),\
        '{:10.6f}'.format(Summary.coefficientStandardErrors[i]),\
        '{:8.3f}'.format(Summary.tValues[i]),\
        '{:10.6f}'.format(Summary.pValues[i]))

    print ("##",'---')
#     print ("##","Mean squared error: % .6f" \
#             % Summary.meanSquaredError, ", RMSE: % .6f" \
#             % Summary.rootMeanSquaredError )
#     print ("##","Multiple R-squared: %f" % Summary.r2, ", \
#              Total iterations: %i"% Summary.totalIterations)
```

```
modelsummary(model.stages[-1])
```

You will get the following summary results:

```
Note: the last rows are the information for Intercept
('##', '-------------------------------------------------')
('##', '  Estimate   |   Std.Error | t Values  |  P-value')
('##', '  0.042857', '  0.001668', '  25.692', '  0.000000')
('##', '  0.199922', '  0.009881', '  20.232', '  0.000000')
('##', ' -0.001957', '  0.006917', '  -0.283', '  0.777757')
('##', '  3.007515', '  0.406389', '   7.401', '  0.000000')
('##', '---')
```

10. Make predictions

```
# Make predictions.
predictions = model.transform(testData)
```

```
# Select example rows to display.
predictions.select("features","label","predictedLabel").show(5)
```

```
+---------------+-----+------------------+
|       features|label|        prediction|
+---------------+-----+------------------+
| [0.7,39.6,8.7]|  1.6|10.937383732327625|
| [4.1,11.6,5.7]|  3.2| 5.491166258750164|
|[7.3,28.1,41.4]|  5.5|   8.8571603947873|
|  [8.6,2.1,1.0]|  4.8| 3.793966281660073|
|[17.2,4.1,31.6]|  5.9| 4.502507124763654|
+---------------+-----+------------------+
only showing top 5 rows
```

11. Evaluation

```
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.evaluation import RegressionEvaluator
# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(labelCol="label",
                                predictionCol="prediction",
                                metricName="rmse")

rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

The final Root Mean Squared Error (RMSE) is as follows:

```
Root Mean Squared Error (RMSE) on test data = 1.89857
```

```
y_true = predictions.select("label").toPandas()
y_pred = predictions.select("prediction").toPandas()

import sklearn.metrics
r2_score = sklearn.metrics.r2_score(y_true, y_pred)
print('r2_score: {0}'.format(r2_score))
```

Then you will get the R^2 value:

```
r2_score: 0.87707391843
```

9.3 Decision tree Regression

9.3.1 Introduction

9.3.2 How to solve it?

9.3.3 Demo

- The Jupyter notebook can be download from Decision Tree Regression.
- For more details about the parameters, please visit Decision Tree Regressor API .

1. Set up spark context and SparkSession

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark regression example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

2. Load dataset

```
df = spark.read.format('com.databricks.spark.csv').\
                        options(header='true', \
                        inferschema='true').\
            load("../data/Advertising.csv",header=True);
```

check the data set

```
df.show(5,True)
df.printSchema()
```

Then you will get

```
+-----+-----+---------+-----+
|   TV|Radio|Newspaper|Sales|
+-----+-----+---------+-----+
|230.1| 37.8|     69.2| 22.1|
| 44.5| 39.3|     45.1| 10.4|
| 17.2| 45.9|     69.3|  9.3|
|151.5| 41.3|     58.5| 18.5|
|180.8| 10.8|     58.4| 12.9|
+-----+-----+---------+-----+
only showing top 5 rows

root
 |-- TV: double (nullable = true)
 |-- Radio: double (nullable = true)
 |-- Newspaper: double (nullable = true)
 |-- Sales: double (nullable = true)
```

You can also get the Statistical resutls from the data frame (Unfortunately, it only works for numerical).

```
df.describe().show()
```

Then you will get

```
+-------+-----------------+------------------+------------------+------------------+
↪-------+
|summary|               TV|             Radio|         Newspaper|
↪Sales|
+-------+-----------------+------------------+------------------+------------------+
↪-------+
|  count|              200|               200|               200|
↪  200|
|   mean|         147.0425|23.264000000000024|30.553999999999995|14.
↪022500000000003|
| stddev|85.85423631490805|14.846809176168728| 21.77862083852283| 5.
↪217456565710477|
|    min|              0.7|               0.0|               0.3|
↪  1.6|
|    max|            296.4|              49.6|             114.0|
↪ 27.0|
+-------+-----------------+------------------+------------------+------------------+
↪-------+
```

(continues on next page)

(continued from previous page)

3. Convert the data to dense vector (**features** and **label**)

Note:

You are strongly encouraged to try my `get_dummy` function for dealing with the categorical data in comple dataset.

Supervised learning version:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,
↪labelCol):

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer,␣
↪OneHotEncoder, VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
↪indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
↪getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
↪getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol=
↪"features")

    pipeline = Pipeline(stages=indexers + encoders +␣
↪[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label',col(labelCol))

    return data.select(indexCol,'features','label')
```

Unsupervised learning version:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols):
    '''
    Get dummy variables and concat with continuous variables␣
↪for unsupervised learning.
```

(continues on next page)

(continued from previous page)

```
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical␣
↪data
    :param continuousCols:  the name list of the numerical␣
↪data
    :return k: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com
    '''

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
↪indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
↪getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
↪getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol=
↪"features")

    pipeline = Pipeline(stages=indexers + encoders +␣
↪[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    return data.select(indexCol,'features')
```

Two in one:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol,
↪dropLast=False):

    '''
    Get dummy variables and concat with continuous variables for ml␣
↪modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols:  the name list of the numerical data
    :param labelCol:  the name of label column
    :param dropLast:  the flag of drop last column
    :return: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com
```

(continues on next page)

(continued from previous page)

```
    >>> df = spark.createDataFrame([
                    (0, "a"),
                    (1, "b"),
                    (2, "c"),
                    (3, "a"),
                    (4, "a"),
                    (5, "c")
                ], ["id", "category"])

    >>> indexCol = 'id'
    >>> categoricalCols = ['category']
    >>> continuousCols = []
    >>> labelCol = []

    >>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,
↪labelCol)
    >>> mat.show()

    >>>
        +---+-------------+
        | id|     features|
        +---+-------------+
        |  0|[1.0,0.0,0.0]|
        |  1|[0.0,0.0,1.0]|
        |  2|[0.0,1.0,0.0]|
        |  3|[1.0,0.0,0.0]|
        |  4|[1.0,0.0,0.0]|
        |  5|[0.0,1.0,0.0]|
        +---+-------------+
    '''

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder,␣
↪VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".
↪format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()),dropLast=dropLast)
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol()␣
↪for encoder in encoders]
                                + continuousCols, outputCol="features
↪")
```

(continues on next page)

(continued from previous page)

```
    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    if indexCol and labelCol:
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
        return data.select(indexCol,'features','label')
    elif not indexCol and labelCol:
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
        return data.select('features','label')
    elif indexCol and not labelCol:
        # for unsupervised learning
        return data.select(indexCol,'features')
    elif not indexCol and not labelCol:
        # for unsupervised learning
        return data.select('features')
```

```
from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

# I provide two ways to build the features and labels

# method 1 (good for small feature):
#def transData(row):
#    return Row(label=row["Sales"],
#               features=Vectors.dense([row["TV"],
#                                       row["Radio"],
#                                       row["Newspaper"]]))

# Method 2 (good for large features):
def transData(data):
return data.rdd.map(lambda r: [Vectors.dense(r[:-1]),r[-1]]).toDF(['features',
↪'label'])
```

```
transformed= transData(df)
transformed.show(5)
```

```
+-----------------+-----+
|         features|label|
+-----------------+-----+
|[230.1,37.8,69.2]| 22.1|
| [44.5,39.3,45.1]| 10.4|
| [17.2,45.9,69.3]|  9.3|
|[151.5,41.3,58.5]| 18.5|
|[180.8,10.8,58.4]| 12.9|
+-----------------+-----+
only showing top 5 rows
```

Note: You will find out that all of the machine learning algorithms in Spark are based on the **features** and **label**. That is to say, you can play with all of the machine learning algorithms in Spark when you get ready the **features** and **label**.

4. Convert the data to dense vector

```
# convert the data to dense vector
def transData(data):
    return data.rdd.map(lambda r: [r[-1], Vectors.dense(r[:-1])]).\
           toDF(['label','features'])

transformed = transData(df)
transformed.show(5)
```

5. Deal with the Categorical variables

```
from pyspark.ml import Pipeline
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

# Automatically identify categorical features, and index them.
# We specify maxCategories so features with > 4
# distinct values are treated as continuous.

featureIndexer = VectorIndexer(inputCol="features", \
                               outputCol="indexedFeatures",\
                               maxCategories=4).fit(transformed)

data = featureIndexer.transform(transformed)
```

When you check you data at this point, you will get

```
+-----------------+-----+-----------------+
|         features|label|  indexedFeatures|
+-----------------+-----+-----------------+
|[230.1,37.8,69.2]| 22.1|[230.1,37.8,69.2]|
| [44.5,39.3,45.1]| 10.4| [44.5,39.3,45.1]|
| [17.2,45.9,69.3]|  9.3| [17.2,45.9,69.3]|
|[151.5,41.3,58.5]| 18.5|[151.5,41.3,58.5]|
|[180.8,10.8,58.4]| 12.9|[180.8,10.8,58.4]|
+-----------------+-----+-----------------+
only showing top 5 rows
```

6. Split the data into training and test sets (40% held out for testing)

```
# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = transformed.randomSplit([0.6, 0.4])
```

You can check your train and test data as follows (In my opinion, it is always to good to keep tracking your data during prototype pahse):

```
trainingData.show(5)
testData.show(5)
```

Then you will get

```
+--------------+-----+--------------+
|      features|label|indexedFeatures|
+--------------+-----+--------------+
| [4.1,11.6,5.7]|  3.2| [4.1,11.6,5.7]|
|[7.3,28.1,41.4]|  5.5|[7.3,28.1,41.4]|
| [8.4,27.2,2.1]|  5.7| [8.4,27.2,2.1]|
|  [8.6,2.1,1.0]|  4.8|  [8.6,2.1,1.0]|
|[8.7,48.9,75.0]|  7.2|[8.7,48.9,75.0]|
+--------------+-----+--------------+
only showing top 5 rows

+---------------+-----+---------------+
|       features|label|indexedFeatures|
+---------------+-----+---------------+
|  [0.7,39.6,8.7]|  1.6|  [0.7,39.6,8.7]|
|  [5.4,29.9,9.4]|  5.3|  [5.4,29.9,9.4]|
| [7.8,38.9,50.6]|  6.6| [7.8,38.9,50.6]|
|[17.2,45.9,69.3]|  9.3|[17.2,45.9,69.3]|
|[18.7,12.1,23.4]|  6.7|[18.7,12.1,23.4]|
+---------------+-----+---------------+
only showing top 5 rows
```

7. Fit Decision Tree Regression Model

```
from pyspark.ml.regression import DecisionTreeRegressor

# Train a DecisionTree model.
dt = DecisionTreeRegressor(featuresCol="indexedFeatures")
```

8. Pipeline Architecture

```
# Chain indexer and tree in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, dt])

model = pipeline.fit(trainingData)
```

9. Make predictions

```
# Make predictions.
predictions = model.transform(testData)
```

```
# Select example rows to display.
predictions.select("features","label","predictedLabel").show(5)
```

```
+----------+-----+---------------+
|prediction|label|       features|
```

(continues on next page)

(continued from previous page)

```
+----------+-----+----------------+
|       7.2|  1.6|  [0.7,39.6,8.7]|
|       7.3|  5.3|  [5.4,29.9,9.4]|
|       7.2|  6.6| [7.8,38.9,50.6]|
|      8.64|  9.3|[17.2,45.9,69.3]|
|      6.45|  6.7|[18.7,12.1,23.4]|
+----------+-----+----------------+
only showing top 5 rows
```

10. Evaluation

```
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.evaluation import RegressionEvaluator
# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(labelCol="label",
                                predictionCol="prediction",
                                metricName="rmse")

rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

The final Root Mean Squared Error (RMSE) is as follows:

```
Root Mean Squared Error (RMSE) on test data = 1.50999
```

```
y_true = predictions.select("label").toPandas()
y_pred = predictions.select("prediction").toPandas()

import sklearn.metrics
r2_score = sklearn.metrics.r2_score(y_true, y_pred)
print('r2_score: {0}'.format(r2_score))
```

Then you will get the R^2 value:

```
r2_score: 0.911024318967
```

You may also check the importance of the features:

```
model.stages[1].featureImportances
```

The you will get the weight for each features

```
SparseVector(3, {0: 0.6811, 1: 0.3187, 2: 0.0002})
```

9.4 Random Forest Regression

9.4.1 Introduction

9.4.2 How to solve it?

9.4.3 Demo

- The Jupyter notebook can be download from Random Forest Regression.
- For more details about the parameters, please visit Random Forest Regressor API .

1. Set up spark context and SparkSession

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark RandomForest Regression example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

2. Load dataset

```
df = spark.read.format('com.databricks.spark.csv').\
                                options(header='true', \
                                inferschema='true').\
                                load("../data/Advertising.csv",header=True);

df.show(5,True)
df.printSchema()
```

```
+-----+-----+---------+-----+
|   TV|Radio|Newspaper|Sales|
+-----+-----+---------+-----+
|230.1| 37.8|     69.2| 22.1|
| 44.5| 39.3|     45.1| 10.4|
| 17.2| 45.9|     69.3|  9.3|
|151.5| 41.3|     58.5| 18.5|
|180.8| 10.8|     58.4| 12.9|
+-----+-----+---------+-----+
only showing top 5 rows

root
 |-- TV: double (nullable = true)
 |-- Radio: double (nullable = true)
 |-- Newspaper: double (nullable = true)
 |-- Sales: double (nullable = true)
```

```
df.describe().show()
```

(continues on next page)

(continued from previous page)

```
+-------+-----------------+------------------+------------------+------------
↪-----+
|summary|               TV|             Radio|         Newspaper|
↪Sales|
+-------+-----------------+------------------+------------------+------------
↪-----+
|  count|              200|               200|               200|
↪  200|
|   mean|         147.0425|23.264000000000024|30.553999999999995|14.
↪022500000000003|
| stddev|85.85423631490805|14.846809176168728| 21.77862083852283| 5.
↪217456565710477|
|    min|              0.7|               0.0|               0.3|
↪  1.6|
|    max|            296.4|              49.6|             114.0|
↪ 27.0|
+-------+-----------------+------------------+------------------+------------
↪-----+
```

3. Convert the data to dense vector (**features** and **label**)

Note:

You are strongly encouraged to try my `get_dummy` function for dealing with the categorical data in comple dataset.

Supervised learning version:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,
↪labelCol):

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer,
↪OneHotEncoder, VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
↪indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
↪getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
↪getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol=
↪"features")
```

(continues on next page)

(continued from previous page)

```
    pipeline = Pipeline(stages=indexers + encoders +␣
↪[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label',col(labelCol))

    return data.select(indexCol,'features','label')
```

Unsupervised learning version:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols):
    '''
    Get dummy variables and concat with continuous variables␣
↪for unsupervised learning.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical␣
↪data
    :param continuousCols:  the name list of the numerical␣
↪data
    :return k: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com
    '''

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
↪indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
↪getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
↪getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol=
↪"features")

    pipeline = Pipeline(stages=indexers + encoders +␣
↪[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    return data.select(indexCol,'features')
```

Two in one:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol,
↪dropLast=False):

    '''
    Get dummy variables and concat with continuous variables for ml␣
↪modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols:  the name list of the numerical data
    :param labelCol:  the name of label column
    :param dropLast:  the flag of drop last column
    :return: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com

    >>> df = spark.createDataFrame([
                    (0, "a"),
                    (1, "b"),
                    (2, "c"),
                    (3, "a"),
                    (4, "a"),
                    (5, "c")
                ], ["id", "category"])

    >>> indexCol = 'id'
    >>> categoricalCols = ['category']
    >>> continuousCols = []
    >>> labelCol = []

    >>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,
↪labelCol)
    >>> mat.show()

    >>>
        +---+-------------+
        | id|     features|
        +---+-------------+
        |  0|[1.0,0.0,0.0]|
        |  1|[0.0,0.0,1.0]|
        |  2|[0.0,1.0,0.0]|
        |  3|[1.0,0.0,0.0]|
        |  4|[1.0,0.0,0.0]|
        |  5|[0.0,1.0,0.0]|
        +---+-------------+
    '''

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder,␣
↪VectorAssembler
    from pyspark.sql.functions import col
```

(continues on next page)

(continued from previous page)

```
    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".
↪format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()),dropLast=dropLast)
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol()␣
↪for encoder in encoders]
                                + continuousCols, outputCol="features
↪")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    if indexCol and labelCol:
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
        return data.select(indexCol,'features','label')
    elif not indexCol and labelCol:
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
        return data.select('features','label')
    elif indexCol and not labelCol:
        # for unsupervised learning
        return data.select(indexCol,'features')
    elif not indexCol and not labelCol:
        # for unsupervised learning
        return data.select('features')
```

```
from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

# convert the data to dense vector
#def transData(row):
#    return Row(label=row["Sales"],
#               features=Vectors.dense([row["TV"],
#                                       row["Radio"],
#                                       row["Newspaper"]]))
def transData(data):
    return data.rdd.map(lambda r: [Vectors.dense(r[:-1]),r[-1]]).toDF([
↪'features','label'])
```

4. Convert the data to dense vector

```
transformed= transData(df)
transformed.show(5)
```

```
+-----------------+-----+
|         features|label|
+-----------------+-----+
|[230.1,37.8,69.2]| 22.1|
| [44.5,39.3,45.1]| 10.4|
| [17.2,45.9,69.3]|  9.3|
|[151.5,41.3,58.5]| 18.5|
|[180.8,10.8,58.4]| 12.9|
+-----------------+-----+
only showing top 5 rows
```

5. Deal with the Categorical variables

```
from pyspark.ml import Pipeline
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

featureIndexer = VectorIndexer(inputCol="features", \
                               outputCol="indexedFeatures",\
                               maxCategories=4).fit(transformed)

data = featureIndexer.transform(transformed)
data.show(5,True)
```

```
+-----------------+-----+-----------------+
|         features|label|  indexedFeatures|
+-----------------+-----+-----------------+
|[230.1,37.8,69.2]| 22.1|[230.1,37.8,69.2]|
| [44.5,39.3,45.1]| 10.4| [44.5,39.3,45.1]|
| [17.2,45.9,69.3]|  9.3| [17.2,45.9,69.3]|
|[151.5,41.3,58.5]| 18.5|[151.5,41.3,58.5]|
|[180.8,10.8,58.4]| 12.9|[180.8,10.8,58.4]|
+-----------------+-----+-----------------+
only showing top 5 rows
```

6. Split the data into training and test sets (40% held out for testing)

```
# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = data.randomSplit([0.6, 0.4])

trainingData.show(5)
testData.show(5)
```

```
+---------------+-----+---------------+
|        features|label| indexedFeatures|
+---------------+-----+---------------+
|  [0.7,39.6,8.7]|  1.6|  [0.7,39.6,8.7]|
```

(continues on next page)

(continued from previous page)

```
|   [8.6,2.1,1.0]|  4.8|   [8.6,2.1,1.0]|
| [8.7,48.9,75.0]|  7.2| [8.7,48.9,75.0]|
|[11.7,36.9,45.2]|  7.3|[11.7,36.9,45.2]|
|[13.2,15.9,49.6]|  5.6|[13.2,15.9,49.6]|
+----------------+-----+----------------+
only showing top 5 rows

+---------------+-----+---------------+
|       features|label|indexedFeatures|
+---------------+-----+---------------+
| [4.1,11.6,5.7]|  3.2| [4.1,11.6,5.7]|
| [5.4,29.9,9.4]|  5.3| [5.4,29.9,9.4]|
|[7.3,28.1,41.4]|  5.5|[7.3,28.1,41.4]|
|[7.8,38.9,50.6]|  6.6|[7.8,38.9,50.6]|
| [8.4,27.2,2.1]|  5.7| [8.4,27.2,2.1]|
+---------------+-----+---------------+
only showing top 5 rows
```

7. Fit RandomForest Regression Model

```
# Import LinearRegression class
from pyspark.ml.regression import RandomForestRegressor

# Define LinearRegression algorithm
rf = RandomForestRegressor() # featuresCol="indexedFeatures",numTrees=2,␣
↪maxDepth=2, seed=42
```

Note: If you decide to use the `indexedFeatures` features, you need to add the parameter `featuresCol="indexedFeatures"`.

8. Pipeline Architecture

```
# Chain indexer and tree in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, rf])
model = pipeline.fit(trainingData)
```

9. Make predictions

```
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("features","label", "prediction").show(5)
```

```
+---------------+-----+------------------+
|       features|label|        prediction|
+---------------+-----+------------------+
| [4.1,11.6,5.7]|  3.2| 8.155439814814816|
| [5.4,29.9,9.4]|  5.3|10.412769901394899|
|[7.3,28.1,41.4]|  5.5| 12.13735648148148|
```

(continues on next page)

(continued from previous page)

```
|[7.8,38.9,50.6]|  6.6|11.321796703296704|
| [8.4,27.2,2.1]|  5.7|12.071421957671957|
+---------------+-----+------------------+
only showing top 5 rows
```

10. Evaluation

```
# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

```
Root Mean Squared Error (RMSE) on test data = 2.35912
```

```
import sklearn.metrics
r2_score = sklearn.metrics.r2_score(y_true, y_pred)
print('r2_score: {:4.3f}'.format(r2_score))
```

```
r2_score: 0.831
```

11. Feature importances

```
model.stages[-1].featureImportances
```

```
SparseVector(3, {0: 0.4994, 1: 0.3196, 2: 0.181})
```

```
model.stages[-1].trees
```

```
[DecisionTreeRegressionModel (uid=dtr_c75f1c75442c) of depth 5 with 43 nodes,
 DecisionTreeRegressionModel (uid=dtr_70fc2d441581) of depth 5 with 45 nodes,
 DecisionTreeRegressionModel (uid=dtr_bc8464f545a7) of depth 5 with 31 nodes,
 DecisionTreeRegressionModel (uid=dtr_a8a7e5367154) of depth 5 with 59 nodes,
 DecisionTreeRegressionModel (uid=dtr_3ea01314fcbc) of depth 5 with 47 nodes,
 DecisionTreeRegressionModel (uid=dtr_be9a04ac22a6) of depth 5 with 45 nodes,
 DecisionTreeRegressionModel (uid=dtr_38610d47328a) of depth 5 with 51 nodes,
 DecisionTreeRegressionModel (uid=dtr_bf14aea0ad3b) of depth 5 with 49 nodes,
 DecisionTreeRegressionModel (uid=dtr_cde24ebd6bb6) of depth 5 with 39 nodes,
 DecisionTreeRegressionModel (uid=dtr_a1fc9bd4fbeb) of depth 5 with 57 nodes,
 DecisionTreeRegressionModel (uid=dtr_37798d6db1ba) of depth 5 with 41 nodes,
 DecisionTreeRegressionModel (uid=dtr_c078b73ada63) of depth 5 with 41 nodes,
 DecisionTreeRegressionModel (uid=dtr_fd00e3a070ad) of depth 5 with 55 nodes,
 DecisionTreeRegressionModel (uid=dtr_9d01d5fb8604) of depth 5 with 45 nodes,
 DecisionTreeRegressionModel (uid=dtr_8bd8bdddf642) of depth 5 with 41 nodes,
 DecisionTreeRegressionModel (uid=dtr_e53b7bae30f8) of depth 5 with 49 nodes,
 DecisionTreeRegressionModel (uid=dtr_808a869db21c) of depth 5 with 47 nodes,
 DecisionTreeRegressionModel (uid=dtr_64d0916bceb0) of depth 5 with 33 nodes,
 DecisionTreeRegressionModel (uid=dtr_0891055fff94) of depth 5 with 55 nodes,
 DecisionTreeRegressionModel (uid=dtr_19c8bbad26c2) of depth 5 with 51 nodes]
```

9.5 Gradient-boosted tree regression

9.5.1 Introduction

9.5.2 How to solve it?

9.5.3 Demo

- The Jupyter notebook can be download from Gradient-boosted tree regression.
- For more details about the parameters, please visit Gradient boosted tree API .

1. Set up spark context and SparkSession

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark GBTRegressor example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

2. Load dataset

```
df = spark.read.format('com.databricks.spark.csv').\
                                options(header='true', \
                                inferschema='true').\
                                load("../data/Advertising.csv",header=True);

df.show(5,True)
df.printSchema()
```

```
+-----+-----+---------+-----+
|   TV|Radio|Newspaper|Sales|
+-----+-----+---------+-----+
|230.1| 37.8|     69.2| 22.1|
| 44.5| 39.3|     45.1| 10.4|
| 17.2| 45.9|     69.3|  9.3|
|151.5| 41.3|     58.5| 18.5|
|180.8| 10.8|     58.4| 12.9|
+-----+-----+---------+-----+
only showing top 5 rows

root
 |-- TV: double (nullable = true)
 |-- Radio: double (nullable = true)
 |-- Newspaper: double (nullable = true)
 |-- Sales: double (nullable = true)
```

```
df.describe().show()
```

(continues on next page)

(continued from previous page)

```
+-------+-----------------+------------------+------------------+------------
↪-----+
|summary|               TV|             Radio|         Newspaper|
↪Sales|
+-------+-----------------+------------------+------------------+------------
↪-----+
|  count|              200|               200|               200|
↪  200|
|   mean|         147.0425|23.264000000000024|30.553999999999995|14.
↪022500000000003|
| stddev|85.85423631490805|14.846809176168728| 21.77862083852283| 5.
↪217456565710477|
|    min|              0.7|               0.0|               0.3|
↪  1.6|
|    max|            296.4|              49.6|             114.0|
↪ 27.0|
+-------+-----------------+------------------+------------------+------------
↪-----+
```

3. Convert the data to dense vector (**features** and **label**)

Note:

You are strongly encouraged to try my `get_dummy` function for dealing with the categorical data in comple dataset.

Supervised learning version:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,
↪labelCol):

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer,
↪OneHotEncoder, VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
↪indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
↪getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
↪getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol=
↪"features")
```

(continues on next page)

(continued from previous page)

```
    pipeline = Pipeline(stages=indexers + encoders +␣
↪[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label',col(labelCol))

    return data.select(indexCol,'features','label')
```

Unsupervised learning version:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols):
    '''
    Get dummy variables and concat with continuous variables␣
↪for unsupervised learning.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical␣
↪data
    :param continuousCols:  the name list of the numerical␣
↪data
    :return k: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com
    '''

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
↪indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
↪getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
↪getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol=
↪"features")

    pipeline = Pipeline(stages=indexers + encoders +␣
↪[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    return data.select(indexCol,'features')
```

Two in one:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol,
↪dropLast=False):

    '''
    Get dummy variables and concat with continuous variables for ml␣
↪modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols:  the name list of the numerical data
    :param labelCol:  the name of label column
    :param dropLast:  the flag of drop last column
    :return: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com

    >>> df = spark.createDataFrame([
                    (0, "a"),
                    (1, "b"),
                    (2, "c"),
                    (3, "a"),
                    (4, "a"),
                    (5, "c")
                ], ["id", "category"])

    >>> indexCol = 'id'
    >>> categoricalCols = ['category']
    >>> continuousCols = []
    >>> labelCol = []

    >>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,
↪labelCol)
    >>> mat.show()

    >>>
        +---+-------------+
        | id|     features|
        +---+-------------+
        |  0|[1.0,0.0,0.0]|
        |  1|[0.0,0.0,1.0]|
        |  2|[0.0,1.0,0.0]|
        |  3|[1.0,0.0,0.0]|
        |  4|[1.0,0.0,0.0]|
        |  5|[0.0,1.0,0.0]|
        +---+-------------+
    '''

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder,␣
↪VectorAssembler
    from pyspark.sql.functions import col
```

(continues on next page)

(continued from previous page)

```
    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".
↪format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()),dropLast=dropLast)
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol()␣
↪for encoder in encoders]
                                + continuousCols, outputCol="features
↪")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    if indexCol and labelCol:
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
        return data.select(indexCol,'features','label')
    elif not indexCol and labelCol:
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
        return data.select('features','label')
    elif indexCol and not labelCol:
        # for unsupervised learning
        return data.select(indexCol,'features')
    elif not indexCol and not labelCol:
        # for unsupervised learning
        return data.select('features')
```

```
from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

# convert the data to dense vector
#def transData(row):
#    return Row(label=row["Sales"],
#               features=Vectors.dense([row["TV"],
#                                       row["Radio"],
#                                       row["Newspaper"]]))
def transData(data):
    return data.rdd.map(lambda r: [Vectors.dense(r[:-1]),r[-1]]).toDF([
↪'features','label'])
```

4. Convert the data to dense vector

```
transformed= transData(df)
transformed.show(5)
```

```
+-----------------+-----+
|         features|label|
+-----------------+-----+
|[230.1,37.8,69.2]| 22.1|
| [44.5,39.3,45.1]| 10.4|
| [17.2,45.9,69.3]|  9.3|
|[151.5,41.3,58.5]| 18.5|
|[180.8,10.8,58.4]| 12.9|
+-----------------+-----+
only showing top 5 rows
```

5. Deal with the Categorical variables

```
from pyspark.ml import Pipeline
from pyspark.ml.regression import GBTRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

featureIndexer = VectorIndexer(inputCol="features", \
                               outputCol="indexedFeatures",\
                               maxCategories=4).fit(transformed)

data = featureIndexer.transform(transformed)
data.show(5,True)
```

```
+-----------------+-----+-----------------+
|         features|label|  indexedFeatures|
+-----------------+-----+-----------------+
|[230.1,37.8,69.2]| 22.1|[230.1,37.8,69.2]|
| [44.5,39.3,45.1]| 10.4| [44.5,39.3,45.1]|
| [17.2,45.9,69.3]|  9.3| [17.2,45.9,69.3]|
|[151.5,41.3,58.5]| 18.5|[151.5,41.3,58.5]|
|[180.8,10.8,58.4]| 12.9|[180.8,10.8,58.4]|
+-----------------+-----+-----------------+
only showing top 5 rows
```

6. Split the data into training and test sets (40% held out for testing)

```
# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = data.randomSplit([0.6, 0.4])

trainingData.show(5)
testData.show(5)
```

```
+---------------+-----+---------------+
|        features|label| indexedFeatures|
+---------------+-----+---------------+
|  [0.7,39.6,8.7]|  1.6|  [0.7,39.6,8.7]|
```

(continues on next page)

(continued from previous page)

```
|   [8.6,2.1,1.0]|  4.8|   [8.6,2.1,1.0]|
| [8.7,48.9,75.0]|  7.2| [8.7,48.9,75.0]|
|[11.7,36.9,45.2]|  7.3|[11.7,36.9,45.2]|
|[13.2,15.9,49.6]|  5.6|[13.2,15.9,49.6]|
+----------------+-----+----------------+
only showing top 5 rows

+---------------+-----+---------------+
|       features|label|indexedFeatures|
+---------------+-----+---------------+
| [4.1,11.6,5.7]|  3.2| [4.1,11.6,5.7]|
| [5.4,29.9,9.4]|  5.3| [5.4,29.9,9.4]|
|[7.3,28.1,41.4]|  5.5|[7.3,28.1,41.4]|
|[7.8,38.9,50.6]|  6.6|[7.8,38.9,50.6]|
| [8.4,27.2,2.1]|  5.7| [8.4,27.2,2.1]|
+---------------+-----+---------------+
only showing top 5 rows
```

7. Fit RandomForest Regression Model

```
# Import LinearRegression class
from pyspark.ml.regression import GBTRegressor

# Define LinearRegression algorithm
rf = GBTRegressor() #numTrees=2, maxDepth=2, seed=42
```

Note: If you decide to use the `indexedFeatures` features, you need to add the parameter `featuresCol="indexedFeatures"`.

8. Pipeline Architecture

```
# Chain indexer and tree in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, rf])
model = pipeline.fit(trainingData)
```

9. Make predictions

```
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("features","label", "prediction").show(5)
```

```
+----------------+-----+------------------+
|        features|label|        prediction|
+----------------+-----+------------------+
| [7.8,38.9,50.6]|  6.6| 6.836040343319862|
|   [8.6,2.1,1.0]|  4.8| 5.652202764688849|
| [8.7,48.9,75.0]|  7.2| 6.908750296855572|
| [13.1,0.4,25.6]|  5.3| 5.784020210692574|
```

(continues on next page)

(continued from previous page)

```
|[19.6,20.1,17.0]|  7.6|6.8678921062629295|
+----------------+-----+------------------+
only showing top 5 rows
```

10. Evaluation

```
# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

```
Root Mean Squared Error (RMSE) on test data = 1.36939
```

```
import sklearn.metrics
r2_score = sklearn.metrics.r2_score(y_true, y_pred)
print('r2_score: {:4.3f}'.format(r2_score))
```

```
r2_score: 0.932
```

11. Feature importances

```
model.stages[-1].featureImportances
```

```
SparseVector(3, {0: 0.3716, 1: 0.3525, 2: 0.2759})
```

```
model.stages[-1].trees
```

```
[DecisionTreeRegressionModel (uid=dtr_7f5cd2ef7cb6) of depth 5 with 61 nodes,
 DecisionTreeRegressionModel (uid=dtr_ef3ab6baeac9) of depth 5 with 39 nodes,
 DecisionTreeRegressionModel (uid=dtr_07c6e3cf3819) of depth 5 with 45 nodes,
 DecisionTreeRegressionModel (uid=dtr_ce724af79a2b) of depth 5 with 47 nodes,
 DecisionTreeRegressionModel (uid=dtr_d149ecc71658) of depth 5 with 55 nodes,
 DecisionTreeRegressionModel (uid=dtr_d3a79bdea516) of depth 5 with 43 nodes,
 DecisionTreeRegressionModel (uid=dtr_7abc1a337844) of depth 5 with 51 nodes,
 DecisionTreeRegressionModel (uid=dtr_480834b46d8f) of depth 5 with 33 nodes,
 DecisionTreeRegressionModel (uid=dtr_0cbd1eaa3874) of depth 5 with 39 nodes,
 DecisionTreeRegressionModel (uid=dtr_8088ac71a204) of depth 5 with 57 nodes,
 DecisionTreeRegressionModel (uid=dtr_2ceb9e8deb45) of depth 5 with 47 nodes,
 DecisionTreeRegressionModel (uid=dtr_cc334e84e9a2) of depth 5 with 57 nodes,
 DecisionTreeRegressionModel (uid=dtr_a665c562929e) of depth 5 with 41 nodes,
 DecisionTreeRegressionModel (uid=dtr_2999b1ffd2dc) of depth 5 with 45 nodes,
 DecisionTreeRegressionModel (uid=dtr_29965cbe8cfc) of depth 5 with 55 nodes,
 DecisionTreeRegressionModel (uid=dtr_731df51bf0ad) of depth 5 with 41 nodes,
 DecisionTreeRegressionModel (uid=dtr_354cf33424da) of depth 5 with 51 nodes,
 DecisionTreeRegressionModel (uid=dtr_4230f200b1c0) of depth 5 with 41 nodes,
 DecisionTreeRegressionModel (uid=dtr_3279cdc1ce1d) of depth 5 with 45 nodes,
 DecisionTreeRegressionModel (uid=dtr_f474a99ff06e) of depth 5 with 55 nodes]
```

CHAPTER

TEN

REGULARIZATION

In mathematics, statistics, and computer science, particularly in the fields of machine learning and inverse problems, regularization is a process of introducing additional information in order to solve an ill-posed problem or to prevent overfitting (Wikipedia Regularization).

Due to the sparsity within our data, our training sets will often be ill-posed (singular). Applying regularization to the regression has many advantages, including:

1. Converting ill-posed problems to well-posed by adding additional information via the penalty parameter λ
2. Preventing overfitting
3. Variable selection and the removal of correlated variables (Glmnet Vignette). The Ridge method shrinks the coefficients of correlated variables while the LASSO method picks one variable and discards the others. The elastic net penalty is a mixture of these two; if variables are correlated in groups then $\alpha = 0.5$ tends to select the groups as in or out. If α is close to 1, the elastic net performs much like the LASSO method and removes any degeneracies and wild behavior caused by extreme correlations.

10.1 Ordinary least squares regression

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^n} \frac{1}{n} \|\mathbf{X}\boldsymbol{\beta} - \boldsymbol{y}\|^2$$

When $\lambda = 0$ (i.e. `regParam` $= 0$), then there is no penalty.

```
LinearRegression(featuresCol="features", labelCol="label", predictionCol=
↪"prediction", maxIter=100,
regParam=0.0, elasticNetParam=0.0, tol=1e-6, fitIntercept=True,␣
↪standardization=True, solver="auto",
weightCol=None, aggregationDepth=2)
```

10.2 Ridge regression

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^n} \frac{1}{n} \|\mathbf{X}\boldsymbol{\beta} - \boldsymbol{y}\|^2 + \lambda \|\boldsymbol{\beta}\|_2^2$$

When $\lambda > 0$ (i.e. `regParam` > 0) and $\alpha = 0$ (i.e. `elasticNetParam` $= 0$) , then the penalty is an L2 penalty.

```
LinearRegression(featuresCol="features", labelCol="label", predictionCol=
↪"prediction", maxIter=100,
regParam=0.1, elasticNetParam=0.0, tol=1e-6, fitIntercept=True,␣
↪standardization=True, solver="auto",
weightCol=None, aggregationDepth=2)
```

10.3 Least Absolute Shrinkage and Selection Operator (LASSO)

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^n} \frac{1}{n} \|\mathbf{X}\boldsymbol{\beta} - \boldsymbol{y}\|^2 + \lambda \|\boldsymbol{\beta}\|_1$$

When $\lambda > 0$ (i.e. `regParam` > 0) and $\alpha = 1$ (i.e. `elasticNetParam` $= 1$), then the penalty is an L1 penalty.

```
LinearRegression(featuresCol="features", labelCol="label", predictionCol=
↪"prediction", maxIter=100,
regParam=0.0, elasticNetParam=0.0, tol=1e-6, fitIntercept=True,␣
↪standardization=True, solver="auto",
weightCol=None, aggregationDepth=2)
```

10.4 Elastic net

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^n} \frac{1}{n} \|\mathbf{X}\boldsymbol{\beta} - \boldsymbol{y}\|^2 + \lambda(\alpha \|\boldsymbol{\beta}\|_1 + (1-\alpha)\|\boldsymbol{\beta}\|_2^2), \alpha \in (0,1)$$

When $\lambda > 0$ (i.e. `regParam` > 0) and `elasticNetParam` $\in (0,1)$ (i.e. $\alpha \in (0,1)$) , then the penalty is an L1 + L2 penalty.

```
LinearRegression(featuresCol="features", labelCol="label", predictionCol=
↪"prediction", maxIter=100,
regParam=0.0, elasticNetParam=0.0, tol=1e-6, fitIntercept=True,␣
↪standardization=True, solver="auto",
weightCol=None, aggregationDepth=2)
```

CHAPTER

ELEVEN

CLASSIFICATION

Chinese proverb

Birds of a feather folock together. – old Chinese proverb

11.1 Binomial logistic regression

11.1.1 Introduction

11.1.2 Demo

- The Jupyter notebook can be download from Logistic Regression.
- For more details, please visit Logistic Regression API .

Note: In this demo, I introduced a new function `get_dummy` to deal with the categorical data. I highly recommend you to use my `get_dummy` function in the other cases. This function will save a lot of time for you.

1. Set up spark context and SparkSession

```
from pyspark.sql import SparkSession

        spark = SparkSession \
            .builder \
            .appName("Python Spark Logistic Regression example") \
            .config("spark.some.config.option", "some-value") \
            .getOrCreate()
```

2. Load dataset

```
df = spark.read.format('com.databricks.spark.csv') \
            .options(header='true', inferschema='true') \
```

(continues on next page)

(continued from previous page)

```
            .load("./data/bank.csv",header=True);
df.drop('day','month','poutcome').show(5)
```

```
+---+-----------+-------+---------+-------+-------+-------+----+-------+-----
↪---+--------+-----+--------+---+
|age|
↪job|marital|education|default|balance|housing|loan|contact|duration|campaign|pdays|previ
↪ y|
+---+-----------+-------+---------+-------+-------+-------+----+-------+-----
↪---+--------+-----+--------+---+
| 58|  management|married| tertiary|     no|   2143|    yes|  no|unknown|
↪261|       1|   -1|       0| no|
| 44|  technician| single|secondary|     no|     29|    yes|  no|unknown|
↪151|       1|   -1|       0| no|
| 33|entrepreneur|married|secondary|     no|      2|    yes| yes|unknown|
↪ 76|       1|   -1|       0| no|
| 47| blue-collar|married|  unknown|     no|   1506|    yes|  no|unknown|
↪ 92|       1|   -1|       0| no|
| 33|     unknown| single|  unknown|     no|      1|     no|  no|unknown|
↪198|       1|   -1|       0| no|
+---+-----------+-------+---------+-------+-------+-------+----+-------+-----
↪---+--------+-----+--------+---+
only showing top 5 rows
```

```
df.printSchema()
```

```
root
 |-- age: integer (nullable = true)
 |-- job: string (nullable = true)
 |-- marital: string (nullable = true)
 |-- education: string (nullable = true)
 |-- default: string (nullable = true)
 |-- balance: integer (nullable = true)
 |-- housing: string (nullable = true)
 |-- loan: string (nullable = true)
 |-- contact: string (nullable = true)
 |-- day: integer (nullable = true)
 |-- month: string (nullable = true)
 |-- duration: integer (nullable = true)
 |-- campaign: integer (nullable = true)
 |-- pdays: integer (nullable = true)
 |-- previous: integer (nullable = true)
 |-- poutcome: string (nullable = true)
 |-- y: string (nullable = true)
```

Note:

You are strongly encouraged to try my `get_dummy` function for dealing with the categorical data in complex dataset.

Supervised learning version:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,
↪labelCol):

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer,␣
↪OneHotEncoder, VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
↪indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
↪getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
↪getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol=
↪"features")

    pipeline = Pipeline(stages=indexers + encoders +␣
↪[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label',col(labelCol))

    return data.select(indexCol,'features','label')
```

Unsupervised learning version:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols):
    '''
    Get dummy variables and concat with continuous variables␣
↪for unsupervised learning.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical␣
↪data
    :param continuousCols:  the name list of the numerical␣
↪data
    :return k: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com
    '''
```

(continues on next page)

(continued from previous page)

```
    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
↪indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
↪getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
↪getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol=
↪"features")

    pipeline = Pipeline(stages=indexers + encoders +␣
↪[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    return data.select(indexCol,'features')
```

Two in one:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol,
↪dropLast=False):

    '''
    Get dummy variables and concat with continuous variables for ml␣
↪modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols:  the name list of the numerical data
    :param labelCol:  the name of label column
    :param dropLast:  the flag of drop last column
    :return: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com

    >>> df = spark.createDataFrame([
                    (0, "a"),
                    (1, "b"),
                    (2, "c"),
                    (3, "a"),
                    (4, "a"),
                    (5, "c")
                ], ["id", "category"])

    >>> indexCol = 'id'
```

(continues on next page)

(continued from previous page)

```
    >>> categoricalCols = ['category']
    >>> continuousCols = []
    >>> labelCol = []

    >>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,
↪labelCol)
    >>> mat.show()

    >>>
        +---+-------------+
        | id|     features|
        +---+-------------+
        |  0|[1.0,0.0,0.0]|
        |  1|[0.0,0.0,1.0]|
        |  2|[0.0,1.0,0.0]|
        |  3|[1.0,0.0,0.0]|
        |  4|[1.0,0.0,0.0]|
        |  5|[0.0,1.0,0.0]|
        +---+-------------+
    '''

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder,␣
↪VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".
↪format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()),dropLast=dropLast)
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol()␣
↪for encoder in encoders]
                                + continuousCols, outputCol="features
↪")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    if indexCol and labelCol:
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
        return data.select(indexCol,'features','label')
    elif not indexCol and labelCol:
        # for supervised learning
```

(continues on next page)

(continued from previous page)

```
        data = data.withColumn('label',col(labelCol))
        return data.select('features','label')
    elif indexCol and not labelCol:
        # for unsupervised learning
        return data.select(indexCol,'features')
    elif not indexCol and not labelCol:
        # for unsupervised learning
        return data.select('features')
```

```
def get_dummy(df,categoricalCols,continuousCols,labelCol):

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder,␣
↪VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder␣
↪in encoders]
                                + continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label',col(labelCol))

    return data.select('features','label')
```

3. Deal with categorical data and Convert the data to dense vector

```
catcols = ['job','marital','education','default',
           'housing','loan','contact','poutcome']

num_cols = ['balance', 'duration','campaign','pdays','previous',]
labelCol = 'y'

data = get_dummy(df,catcols,num_cols,labelCol)
data.show(5)
```

```
+--------------------+-----+
|            features|label|
```

(continues on next page)

(continued from previous page)

```
+--------------------+-----+
|(29,[1,11,14,16,1...|   no|
|(29,[2,12,13,16,1...|   no|
|(29,[7,11,13,16,1...|   no|
|(29,[0,11,16,17,1...|   no|
|(29,[12,16,18,20,...|   no|
+--------------------+-----+
only showing top 5 rows
```

4. Deal with Categorical Label and Variables

```
from pyspark.ml.feature import StringIndexer
# Index labels, adding metadata to the label column
labelIndexer = StringIndexer(inputCol='label',
                             outputCol='indexedLabel').fit(data)
labelIndexer.transform(data).show(5, True)
```

```
+--------------------+-----+------------+
|            features|label|indexedLabel|
+--------------------+-----+------------+
|(29,[1,11,14,16,1...|   no|         0.0|
|(29,[2,12,13,16,1...|   no|         0.0|
|(29,[7,11,13,16,1...|   no|         0.0|
|(29,[0,11,16,17,1...|   no|         0.0|
|(29,[12,16,18,20,...|   no|         0.0|
+--------------------+-----+------------+
only showing top 5 rows
```

```
from pyspark.ml.feature import VectorIndexer
# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as␣
↪continuous.
featureIndexer =VectorIndexer(inputCol="features", \
                              outputCol="indexedFeatures", \
                              maxCategories=4).fit(data)
featureIndexer.transform(data).show(5, True)
```

```
+--------------------+-----+--------------------+
|            features|label|     indexedFeatures|
+--------------------+-----+--------------------+
|(29,[1,11,14,16,1...|   no|(29,[1,11,14,16,1...|
|(29,[2,12,13,16,1...|   no|(29,[2,12,13,16,1...|
|(29,[7,11,13,16,1...|   no|(29,[7,11,13,16,1...|
|(29,[0,11,16,17,1...|   no|(29,[0,11,16,17,1...|
|(29,[12,16,18,20,...|   no|(29,[12,16,18,20,...|
+--------------------+-----+--------------------+
only showing top 5 rows
```

5. Split the data to training and test data sets

```
# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = data.randomSplit([0.6, 0.4])

trainingData.show(5,False)
testData.show(5,False)
```

```
+-----------------------------------------------------------------------------
↪--------------------+-----+
|features
↪                     |label|
+-----------------------------------------------------------------------------
↪--------------------+-----+
|(29,[0,11,13,16,17,18,19,21,24,25,26,27],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,-
↪731.0,401.0,4.0,-1.0])|no   |
|(29,[0,11,13,16,17,18,19,21,24,25,26,27],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,-
↪723.0,112.0,2.0,-1.0])|no   |
|(29,[0,11,13,16,17,18,19,21,24,25,26,27],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,-
↪626.0,205.0,1.0,-1.0])|no   |
|(29,[0,11,13,16,17,18,19,21,24,25,26,27],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,-
↪498.0,357.0,1.0,-1.0])|no   |
|(29,[0,11,13,16,17,18,19,21,24,25,26,27],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,-
↪477.0,473.0,2.0,-1.0])|no   |
+-----------------------------------------------------------------------------
↪--------------------+-----+
only showing top 5 rows

+-----------------------------------------------------------------------------
↪--------------------+-----+
|features
↪                     |label|
+-----------------------------------------------------------------------------
↪--------------------+-----+
|(29,[0,11,13,16,17,18,19,21,24,25,26,27],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,-
↪648.0,280.0,2.0,-1.0])|no   |
|(29,[0,11,13,16,17,18,19,21,24,25,26,27],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,-
↪596.0,147.0,1.0,-1.0])|no   |
|(29,[0,11,13,16,17,18,19,21,24,25,26,27],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,-
↪529.0,416.0,4.0,-1.0])|no   |
|(29,[0,11,13,16,17,18,19,21,24,25,26,27],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,-
↪518.0,46.0,5.0,-1.0]) |no   |
|(29,[0,11,13,16,17,18,19,21,24,25,26,27],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,-
↪470.0,275.0,2.0,-1.0])|no   |
+-----------------------------------------------------------------------------
↪--------------------+-----+
only showing top 5 rows
```

6. Fit Logistic Regression Model

```
from pyspark.ml.classification import LogisticRegression
logr = LogisticRegression(featuresCol='indexedFeatures', labelCol=
↪'indexedLabel')
```

7. Pipeline Architecture

```
# Convert indexed labels back to original labels.
labelConverter = IndexToString(inputCol="prediction", outputCol=
↪"predictedLabel",
                               labels=labelIndexer.labels)
```

```
# Chain indexers and tree in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, logr,
↪labelConverter])
```

```
# Train model.  This also runs the indexers.
model = pipeline.fit(trainingData)
```

8. Make predictions

```
# Make predictions.
predictions = model.transform(testData)
# Select example rows to display.
predictions.select("features","label","predictedLabel").show(5)
```

```
+--------------------+-----+--------------+
|            features|label|predictedLabel|
+--------------------+-----+--------------+
|(29,[0,11,13,16,1...|   no|            no|
|(29,[0,11,13,16,1...|   no|            no|
|(29,[0,11,13,16,1...|   no|            no|
|(29,[0,11,13,16,1...|   no|            no|
|(29,[0,11,13,16,1...|   no|            no|
+--------------------+-----+--------------+
only showing top 5 rows
```

9. Evaluation

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Select (prediction, true label) and compute test error
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy
↪")
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g" % (1.0 - accuracy))
```

```
Test Error = 0.0987688
```

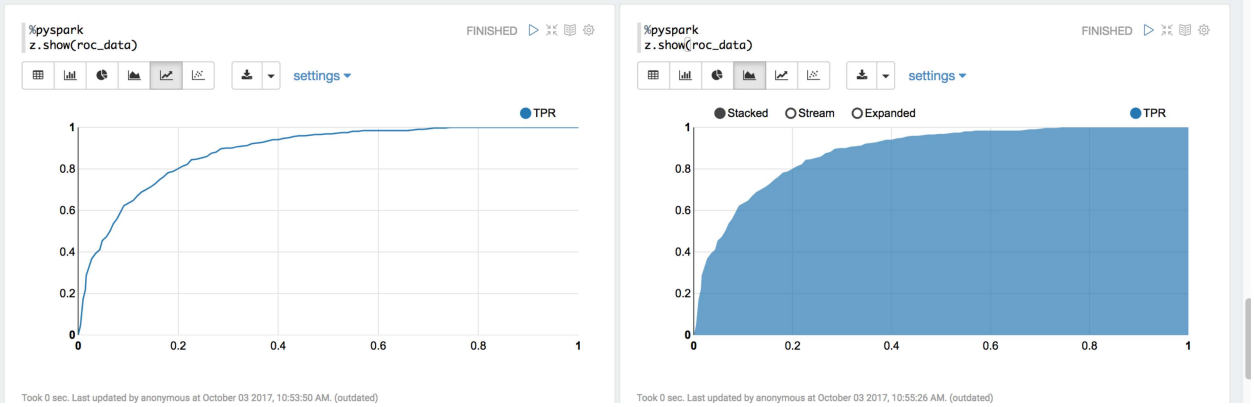
```
lrModel = model.stages[2]
trainingSummary = lrModel.summary

# Obtain the objective per iteration
# objectiveHistory = trainingSummary.objectiveHistory
# print("objectiveHistory:")
# for objective in objectiveHistory:
```

(continues on next page)

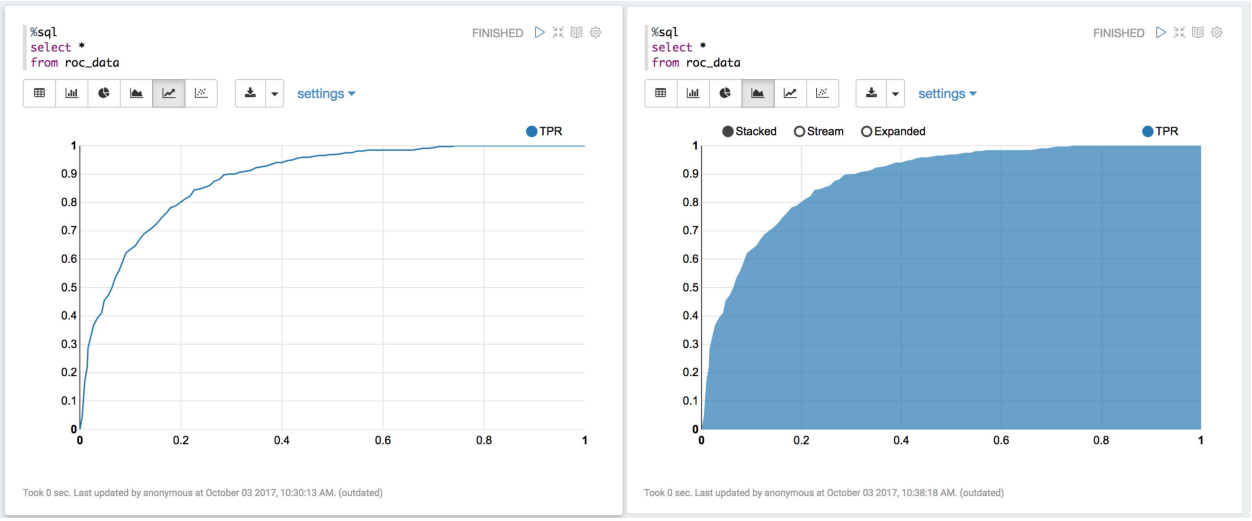
(continued from previous page)

```
#     print(objective)

# Obtain the receiver-operating characteristic as a dataframe and␣
↪areaUnderROC.
trainingSummary.roc.show(5)
print("areaUnderROC: " + str(trainingSummary.areaUnderROC))

# Set the model threshold to maximize F-Measure
fMeasure = trainingSummary.fMeasureByThreshold
maxFMeasure = fMeasure.groupBy().max('F-Measure').select('max(F-Measure)').
↪head(5)
# bestThreshold = fMeasure.where(fMeasure['F-Measure'] == maxFMeasure['max(F-
↪Measure)']) \
#     .select('threshold').head()['threshold']
# lr.setThreshold(bestThreshold)
```

You can use `z.show()` to get the data and plot the ROC curves:



You can also register a TempTable `data.registerTempTable('roc_data')` and then use `sql` to plot the ROC curve:



10. visualization

```
import matplotlib.pyplot as plt
import numpy as np
import itertools

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

```
class_temp = predictions.select("label").groupBy("label")\
                        .count().sort('count', ascending=False).toPandas()
class_temp = class_temp["label"].values.tolist()
class_names = map(str, class_temp)
# # # print(class_name)
class_names
```

```
['no', 'yes']
```

```
from sklearn.metrics import confusion_matrix
y_true = predictions.select("label")
y_true = y_true.toPandas()
```

(continues on next page)

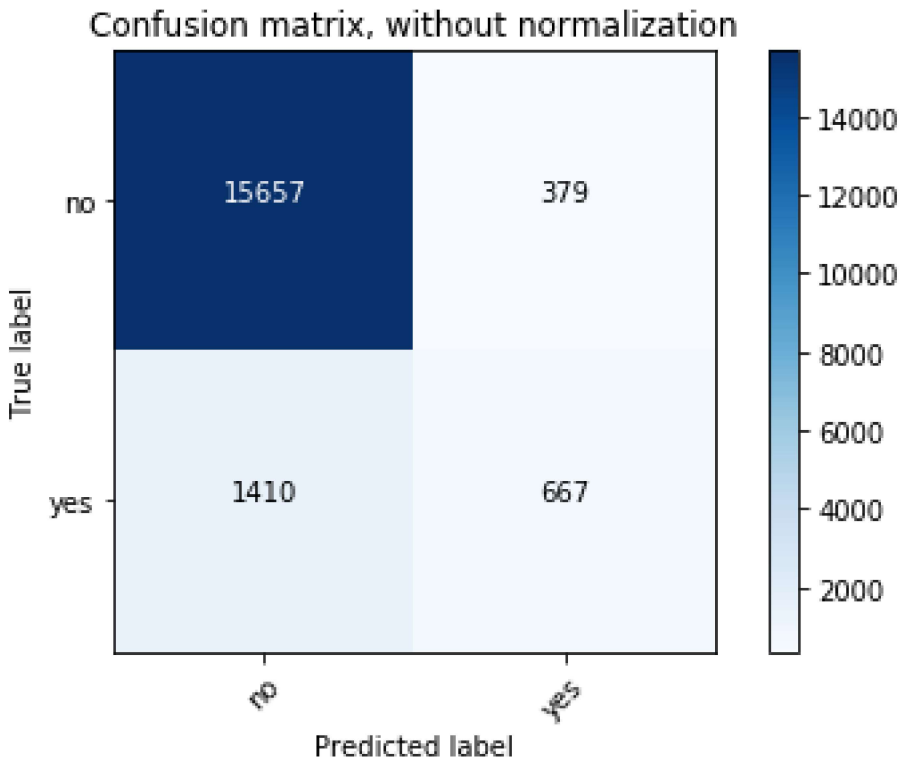
(continued from previous page)

```
y_pred = predictions.select("predictedLabel")
y_pred = y_pred.toPandas()

cnf_matrix = confusion_matrix(y_true, y_pred,labels=class_names)
cnf_matrix
```

```
array([[15657,   379],
       [ 1410,   667]])
```

```
# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names,
                      title='Confusion matrix, without normalization')
plt.show()
```

```
Confusion matrix, without normalization
[[15657   379]
 [ 1410   667]]
```



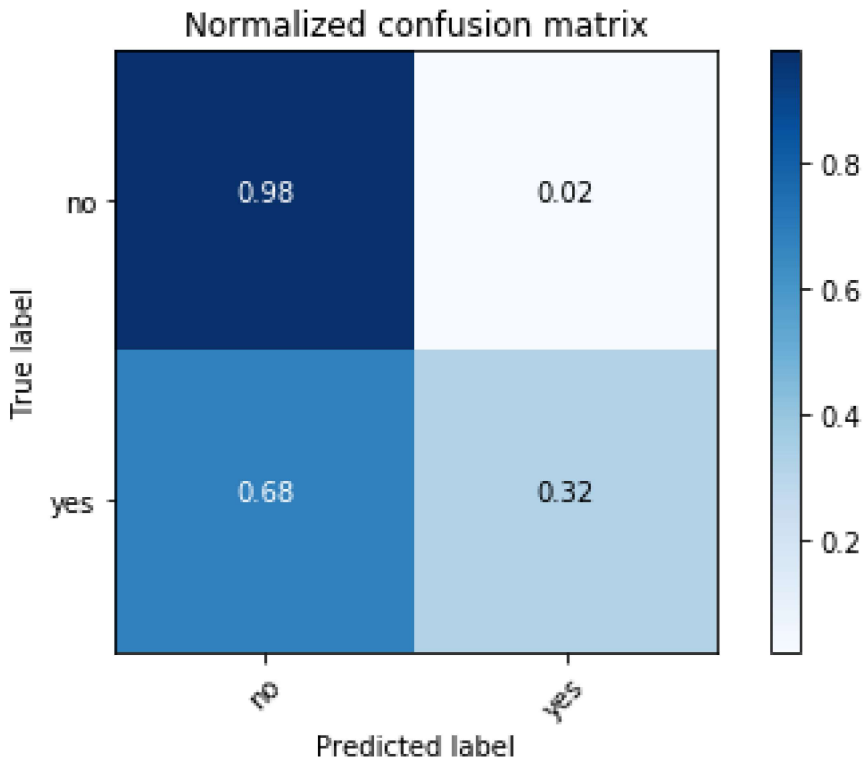

```
# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True,
                      title='Normalized confusion matrix')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

```
Normalized confusion matrix
[[ 0.97636568  0.02363432]
 [ 0.67886375  0.32113625]]
```



11.2 Multinomial logistic regression

11.2.1 Introduction

11.2.2 Demo

- The Jupyter notebook can be download from Logistic Regression.
- For more details, please visit Logistic Regression API .

Note: In this demo, I introduced a new function `get_dummy` to deal with the categorical data. I highly recommend you to use my `get_dummy` function in the other cases. This function will save a lot of time for you.

---

1. Set up spark context and SparkSession

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark MultinomialLogisticRegression classification") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

2. Load dataset

```
df = spark.read.format('com.databricks.spark.csv') \
            .options(header='true', inferschema='true') \
            .load("./data/WineData2.csv",header=True);
df.show(5)
```

```
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----
↪--+-------+
|fixed|volatile|citric|sugar|chlorides|free|total|density| ⌴
↪pH|sulphates|alcohol|quality|
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----
↪--+-------+
|  7.4|     0.7|   0.0|  1.9|    0.076|11.0| 34.0| 0.9978|3.51|     0.56|  ⌴
↪9.4|       5|
|  7.8|    0.88|   0.0|  2.6|    0.098|25.0| 67.0| 0.9968| 3.2|     0.68|  ⌴
↪9.8|       5|
|  7.8|    0.76|  0.04|  2.3|    0.092|15.0| 54.0|  0.997|3.26|     0.65|  ⌴
↪9.8|       5|
| 11.2|    0.28|  0.56|  1.9|    0.075|17.0| 60.0|  0.998|3.16|     0.58|  ⌴
↪9.8|       6|
|  7.4|     0.7|   0.0|  1.9|    0.076|11.0| 34.0| 0.9978|3.51|     0.56|  ⌴
↪9.4|       5|
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----
↪--+-------+
only showing top 5 rows
```

```
df.printSchema()
```

```
root
 |-- fixed: double (nullable = true)
 |-- volatile: double (nullable = true)
 |-- citric: double (nullable = true)
 |-- sugar: double (nullable = true)
 |-- chlorides: double (nullable = true)
 |-- free: double (nullable = true)
 |-- total: double (nullable = true)
 |-- density: double (nullable = true)
 |-- pH: double (nullable = true)
 |-- sulphates: double (nullable = true)
 |-- alcohol: double (nullable = true)
 |-- quality: string (nullable = true)
```

```
# Convert to float format
def string_to_float(x):
    return float(x)

#
def condition(r):
    if (0<= r <= 4):
        label = "low"
    elif(4< r <= 6):
        label = "medium"
    else:
        label = "high"
    return label

from pyspark.sql.functions import udf
from pyspark.sql.types import StringType, DoubleType
string_to_float_udf = udf(string_to_float, DoubleType())
quality_udf = udf(lambda x: condition(x), StringType())

df = df.withColumn("quality", quality_udf("quality"))

df.show(5,True)
```

```
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----
↪--+-------+
|fixed|volatile|citric|sugar|chlorides|free|total|density| ␣
↪pH|sulphates|alcohol|quality|
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----
↪--+-------+
|  7.4|     0.7|   0.0|  1.9|    0.076|11.0| 34.0| 0.9978|3.51|     0.56|   ␣
↪9.4| medium|
|  7.8|    0.88|   0.0|  2.6|    0.098|25.0| 67.0| 0.9968| 3.2|     0.68|   ␣
↪9.8| medium|
|  7.8|    0.76|  0.04|  2.3|    0.092|15.0| 54.0|  0.997|3.26|     0.65|   ␣
↪9.8| medium|
| 11.2|    0.28|  0.56|  1.9|    0.075|17.0| 60.0|  0.998|3.16|     0.58|   ␣
↪9.8| medium|
|  7.4|     0.7|   0.0|  1.9|    0.076|11.0| 34.0| 0.9978|3.51|     0.56|   ␣
↪9.4| medium|
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----
↪--+-------+
only showing top 5 rows
```

```
df.printSchema()
```

```
root
 |-- fixed: double (nullable = true)
 |-- volatile: double (nullable = true)
 |-- citric: double (nullable = true)
 |-- sugar: double (nullable = true)
 |-- chlorides: double (nullable = true)
```

(continues on next page)

(continued from previous page)

```
|-- free: double (nullable = true)
|-- total: double (nullable = true)
|-- density: double (nullable = true)
|-- pH: double (nullable = true)
|-- sulphates: double (nullable = true)
|-- alcohol: double (nullable = true)
|-- quality: string (nullable = true)
```

3. Deal with categorical data and Convert the data to dense vector

---

Note:

You are strongly encouraged to try my `get_dummy` function for dealing with the categorical data in complex dataset.

Supervised learning version:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,
↪labelCol):

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer,␣
↪OneHotEncoder, VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
↪indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
↪getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
↪getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol=
↪"features")

    pipeline = Pipeline(stages=indexers + encoders +␣
↪[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label',col(labelCol))

    return data.select(indexCol,'features','label')
```

Unsupervised learning version:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols):
    '''
    Get dummy variables and concat with continuous variables␣
↪for unsupervised learning.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical␣
↪data
    :param continuousCols:  the name list of the numerical␣
↪data
    :return k: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com
    '''

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
↪indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
↪getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
↪getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol=
↪"features")

    pipeline = Pipeline(stages=indexers + encoders +␣
↪[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    return data.select(indexCol,'features')
```

Two in one:

```
def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol,
↪dropLast=False):

    '''
    Get dummy variables and concat with continuous variables for ml␣
↪modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols:  the name list of the numerical data
    :param labelCol:  the name of label column
    :param dropLast:  the flag of drop last column
```

(continues on next page)

(continued from previous page)

```
    :return: feature matrix

    :author: Wenqiang Feng
    :email:  von198@gmail.com

    >>> df = spark.createDataFrame([
                    (0, "a"),
                    (1, "b"),
                    (2, "c"),
                    (3, "a"),
                    (4, "a"),
                    (5, "c")
                ], ["id", "category"])

    >>> indexCol = 'id'
    >>> categoricalCols = ['category']
    >>> continuousCols = []
    >>> labelCol = []

    >>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,
↪labelCol)
    >>> mat.show()

    >>>
        +---+-------------+
        | id|     features|
        +---+-------------+
        |  0|[1.0,0.0,0.0]|
        |  1|[0.0,0.0,1.0]|
        |  2|[0.0,1.0,0.0]|
        |  3|[1.0,0.0,0.0]|
        |  4|[1.0,0.0,0.0]|
        |  5|[0.0,1.0,0.0]|
        +---+-------------+
    '''

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder,␣
↪VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".
↪format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.
↪getOutputCol()),dropLast=dropLast)
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol()␣
↪for encoder in encoders]
```

(continues on next page)

(continued from previous page)

```
                                    + continuousCols, outputCol="features
↪")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    if indexCol and labelCol:
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
        return data.select(indexCol,'features','label')
    elif not indexCol and labelCol:
        # for supervised learning
        data = data.withColumn('label',col(labelCol))
        return data.select('features','label')
    elif indexCol and not labelCol:
        # for unsupervised learning
        return data.select(indexCol,'features')
    elif not indexCol and not labelCol:
        # for unsupervised learning
        return data.select('features')
```

```
def get_dummy(df,categoricalCols,continuousCols,labelCol):

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder,␣
↪VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
                 for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                 outputCol="{0}_encoded".format(indexer.getOutputCol()))
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder␣
↪in encoders]
                                + continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label',col(labelCol))

    return data.select('features','label')
```

4. Transform the dataset to DataFrame

```
from pyspark.ml.linalg import Vectors # !!!!caution: not from pyspark.mllib.
↪linalg import Vectors
from pyspark.ml import Pipeline
from pyspark.ml.feature import IndexToString,StringIndexer, VectorIndexer
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

def transData(data):
return data.rdd.map(lambda r: [Vectors.dense(r[:-1]),r[-1]]).toDF(['features',
↪'label'])
```

```
transformed = transData(df)
transformed.show(5)
```

```
+--------------------+------+
|            features| label|
+--------------------+------+
|[7.4,0.7,0.0,1.9,...|medium|
|[7.8,0.88,0.0,2.6...|medium|
|[7.8,0.76,0.04,2....|medium|
|[11.2,0.28,0.56,1...|medium|
|[7.4,0.7,0.0,1.9,...|medium|
+--------------------+------+
only showing top 5 rows
```

4. Deal with Categorical Label and Variables

```
# Index labels, adding metadata to the label column
labelIndexer = StringIndexer(inputCol='label',
                             outputCol='indexedLabel').fit(transformed)
labelIndexer.transform(transformed).show(5, True)
```

```
+--------------------+------+------------+
|            features| label|indexedLabel|
+--------------------+------+------------+
|[7.4,0.7,0.0,1.9,...|medium|         0.0|
|[7.8,0.88,0.0,2.6...|medium|         0.0|
|[7.8,0.76,0.04,2....|medium|         0.0|
|[11.2,0.28,0.56,1...|medium|         0.0|
|[7.4,0.7,0.0,1.9,...|medium|         0.0|
+--------------------+------+------------+
only showing top 5 rows
```

```
# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as␣
↪continuous.
featureIndexer =VectorIndexer(inputCol="features", \
                              outputCol="indexedFeatures", \
                              maxCategories=4).fit(transformed)
featureIndexer.transform(transformed).show(5, True)
```

```
+--------------------+------+--------------------+
|            features| label|     indexedFeatures|
+--------------------+------+--------------------+
|[7.4,0.7,0.0,1.9,...|medium|[7.4,0.7,0.0,1.9,...|
|[7.8,0.88,0.0,2.6...|medium|[7.8,0.88,0.0,2.6...|
|[7.8,0.76,0.04,2....|medium|[7.8,0.76,0.04,2....|
|[11.2,0.28,0.56,1...|medium|[11.2,0.28,0.56,1...|
|[7.4,0.7,0.0,1.9,...|medium|[7.4,0.7,0.0,1.9,...|
+--------------------+------+--------------------+
only showing top 5 rows
```

5. Split the data to training and test data sets

```
# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = data.randomSplit([0.6, 0.4])

trainingData.show(5,False)
testData.show(5,False)
```

```
+-----------------------------------------------------------+------+
|features                                                   |label |
+-----------------------------------------------------------+------+
|[4.7,0.6,0.17,2.3,0.058,17.0,106.0,0.9932,3.85,0.6,12.9]   |medium|
|[5.0,0.38,0.01,1.6,0.048,26.0,60.0,0.99084,3.7,0.75,14.0]|medium|
|[5.0,0.4,0.5,4.3,0.046,29.0,80.0,0.9902,3.49,0.66,13.6]   |medium|
|[5.0,0.74,0.0,1.2,0.041,16.0,46.0,0.99258,4.01,0.59,12.5]|medium|
|[5.1,0.42,0.0,1.8,0.044,18.0,88.0,0.99157,3.68,0.73,13.6]|high  |
+-----------------------------------------------------------+------+
only showing top 5 rows

+-----------------------------------------------------------+------+
|features                                                   |label |
+-----------------------------------------------------------+------+
|[4.6,0.52,0.15,2.1,0.054,8.0,65.0,0.9934,3.9,0.56,13.1]    |low   |
|[4.9,0.42,0.0,2.1,0.048,16.0,42.0,0.99154,3.71,0.74,14.0]|high  |
|[5.0,0.42,0.24,2.0,0.06,19.0,50.0,0.9917,3.72,0.74,14.0]  |high  |
|[5.0,1.02,0.04,1.4,0.045,41.0,85.0,0.9938,3.75,0.48,10.5]|low   |
|[5.0,1.04,0.24,1.6,0.05,32.0,96.0,0.9934,3.74,0.62,11.5]  |medium|
+-----------------------------------------------------------+------+
only showing top 5 rows
```

6. Fit Multinomial logisticRegression Classification Model

```
from pyspark.ml.classification import LogisticRegression
logr = LogisticRegression(featuresCol='indexedFeatures', labelCol=
↪'indexedLabel')
```

7. Pipeline Architecture

```
# Convert indexed labels back to original labels.
labelConverter = IndexToString(inputCol="prediction", outputCol=
↪"predictedLabel",
```

(continues on next page)

(continued from previous page)

```
                                    labels=labelIndexer.labels)
```

```
# Chain indexers and tree in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, logr,
↪labelConverter])
```

```
# Train model.  This also runs the indexers.
model = pipeline.fit(trainingData)
```

8. Make predictions

```
# Make predictions.
predictions = model.transform(testData)
# Select example rows to display.
predictions.select("features","label","predictedLabel").show(5)
```

```
+--------------------+------+--------------+
|            features| label|predictedLabel|
+--------------------+------+--------------+
|[4.6,0.52,0.15,2....|   low|        medium|
|[4.9,0.42,0.0,2.1...|  high|          high|
|[5.0,0.42,0.24,2....|  high|          high|
|[5.0,1.02,0.04,1....|   low|        medium|
|[5.0,1.04,0.24,1....|medium|        medium|
+--------------------+------+--------------+
only showing top 5 rows
```

9. Evaluation

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Select (prediction, true label) and compute test error
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy
↪")
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g" % (1.0 - accuracy))
```

```
Test Error = 0.181287
```

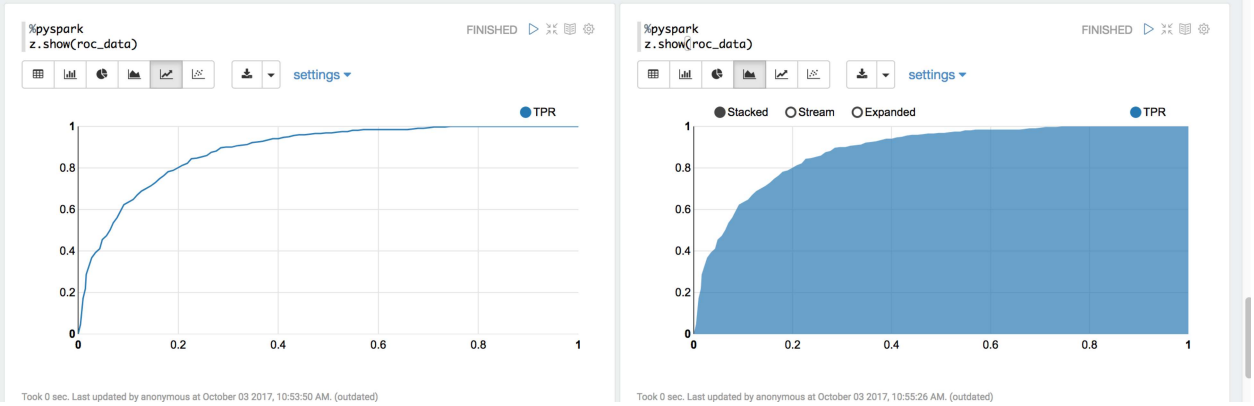
```
lrModel = model.stages[2]
trainingSummary = lrModel.summary

# Obtain the objective per iteration
# objectiveHistory = trainingSummary.objectiveHistory
# print("objectiveHistory:")
# for objective in objectiveHistory:
#     print(objective)
```

(continues on next page)

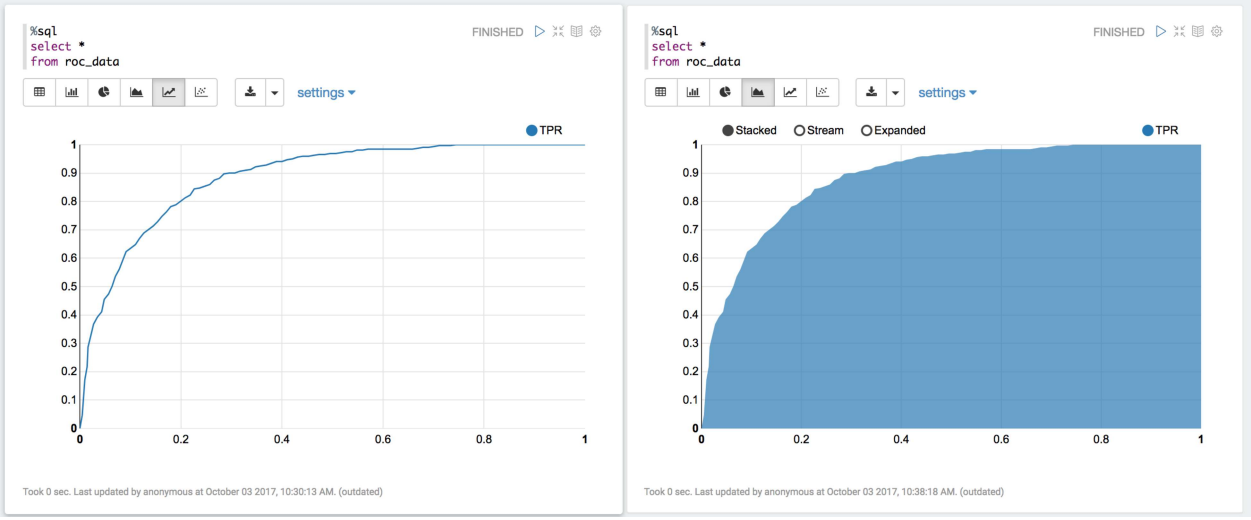
(continued from previous page)

```
# Obtain the receiver-operating characteristic as a dataframe and␣
↪areaUnderROC.
trainingSummary.roc.show(5)
print("areaUnderROC: " + str(trainingSummary.areaUnderROC))

# Set the model threshold to maximize F-Measure
fMeasure = trainingSummary.fMeasureByThreshold
maxFMeasure = fMeasure.groupBy().max('F-Measure').select('max(F-Measure)').
↪head(5)
# bestThreshold = fMeasure.where(fMeasure['F-Measure'] == maxFMeasure['max(F-
↪Measure)']) \
#     .select('threshold').head()['threshold']
# lr.setThreshold(bestThreshold)
```

You can use `z.show()` to get the data and plot the ROC curves:



You can also register a TempTable `data.registerTempTable('roc_data')` and then use `sql` to plot the ROC curve:



10. visualization

```
import matplotlib.pyplot as plt
import numpy as np
import itertools

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

```
class_temp = predictions.select("label").groupBy("label")\
                        .count().sort('count', ascending=False).toPandas()
class_temp = class_temp["label"].values.tolist()
class_names = map(str, class_temp)
# # # print(class_name)
class_names
```

```
['medium', 'high', 'low']
```

```
from sklearn.metrics import confusion_matrix
y_true = predictions.select("label")
y_true = y_true.toPandas()
```

(continues on next page)

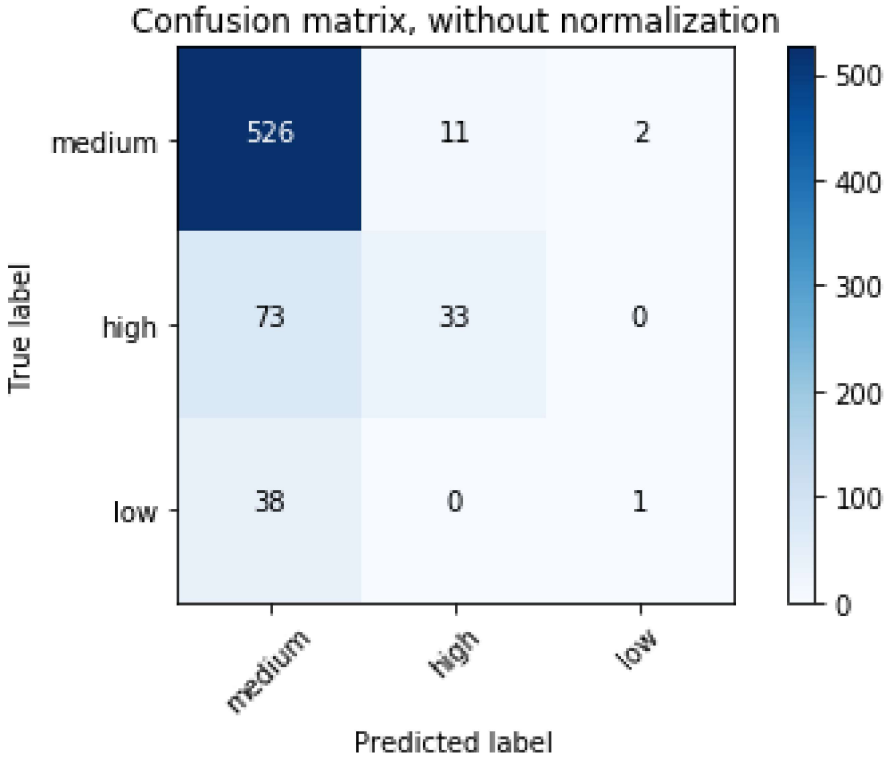
(continued from previous page)

```
y_pred = predictions.select("predictedLabel")
y_pred = y_pred.toPandas()

cnf_matrix = confusion_matrix(y_true, y_pred,labels=class_names)
cnf_matrix
```

```
array([[526,  11,   2],
       [ 73,  33,   0],
       [ 38,   0,   1]])
```

```
# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names,
                      title='Confusion matrix, without normalization')
plt.show()
```

```
Confusion matrix, without normalization
[[526  11   2]
 [ 73  33   0]
 [ 38   0   1]]
```



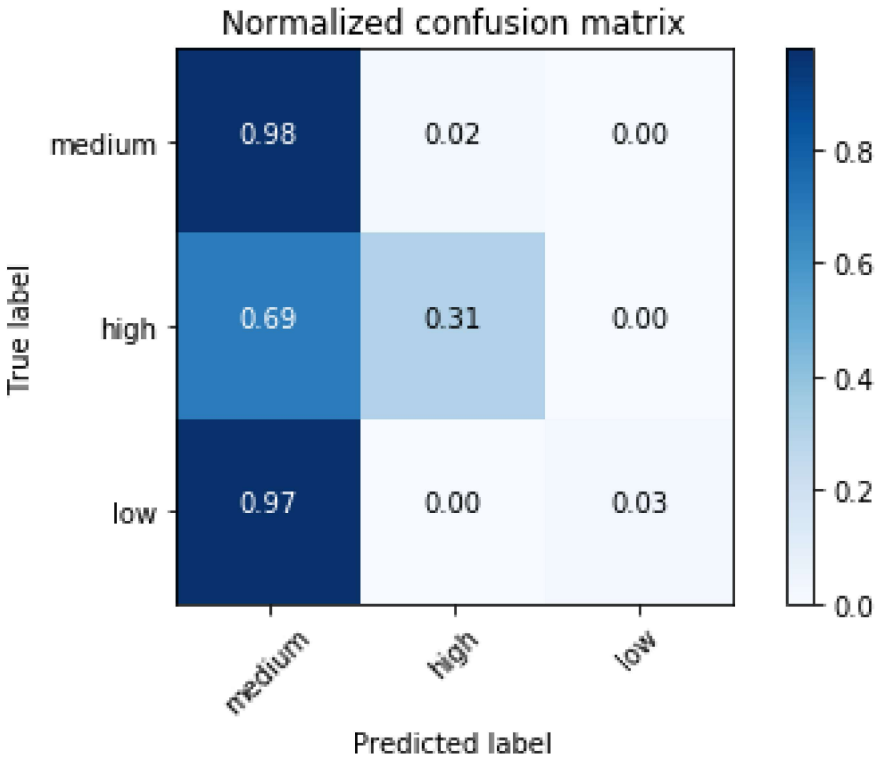

```
# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True,
```

(continues on next page)

(continued from previous page)

```
                      title='Normalized confusion matrix')

plt.show()
```

```
Normalized confusion matrix
[[0.97588126 0.02040816 0.00371058]
 [0.68867925 0.31132075 0.        ]
 [0.97435897 0.         0.02564103]]
```



11.3 Decision tree Classification

11.3.1 Introduction

11.3.2 Demo

- The Jupyter notebook can be download from Decision Tree Classification.
- For more details, please visit DecisionTreeClassifier API .

1. Set up spark context and SparkSession

```
from pyspark.sql import SparkSession
```

(continues on next page)